“十一五”国家重点图书　　计算机科学与技术学科前沿丛书

计算机科学与技术学科研究生系列教材（中文版）

网络安全高级软件编程技术

吴功宜 主编

张建忠 张健 董大凡 许昱玮 等编著



清华大学出版社

“十一五”国家重点图书
计算机科学与技术学科前沿丛书
计算机科学与技术学科研究生系列教材(中文版)

网络安全高级软件编程技术

Advanced Software Programming Technology of Network Security

吴功宜　主编

张建忠　张健　董大凡　许昱玮　等　编著

清华大学出版社
北　京

内 容 简 介

本书的作者队伍是由南开大学计算机系、国家计算机病毒应急处理中心的人员组成。作者在总结多年网络安全科研与教学实践经验的基础上，设计了12个“近似实战”的网络安全软件设计与编程训练的课题。训练课题覆盖了从密码学在网络通信中的应用，网络端口扫描、网络嗅探器、网络诱骗、网络入侵检测、安全Web、防火墙，到Linux内核网络协议栈程序加固、网络病毒与垃圾邮件的检测与防治技术。训练课题接近研究的前沿，覆盖了网络安全研发的主要领域与方向。完成网络安全训练课题的操作系统选择为Linux，完成训练课题不限定任何特殊的硬件环境与编程语言。通过在Linux环境中完成网络安全软件的设计与编程训练，提高读者研发具有自主知识产权的网络安全技术和产品的能力。

本书可以作为计算机、信息安全、软件工程、通信工程、电子信息及相关专业的硕士与工程硕士研究生、博士研究生的教材或参考书，以及本科计算机专业，信息安全专业高年级学生网络安全教材或参考书，也可作为网络安全高级软件编程人才的培训教材与研发工作参考手册。

图书在版编目（CIP）数据

网络安全高级软件编程技术/吴功宜主编；张建忠等编著. —北京：清华大学出版社，2010.4
（计算机科学与技术学科研究生系列教材（中文版））
ISBN 978-7-302-21904-0

Ⅰ. ①网…　Ⅱ. ①吴…　②张…　Ⅲ. ①计算机网络－安全技术－软件设计－高等学校－教材　Ⅳ. ①TP393.08　②TP311.5

中国版本图书馆CIP数据核字（2010）第013245号

责任编辑：张瑞庆　薛　阳
责任校对：时翠兰
责任印制：

出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185×260　**印　张**：25.75　**字　数**：621千字
附光盘1张
版　　次：2010年4月第1版　　**印　次**：2010年4月第1次印刷
印　　数：1～0000
定　　价：00.00元

产品编号：034697-01

序

未来的社会是信息化的社会，计算机科学与技术在其中占据了最重要的地位，这对高素质创新型计算机人才的培养提出了迫切的要求。计算机科学与技术已经成为一门基础技术学科，理论性和技术性都很强。与传统的数学、物理和化学等基础学科相比，该学科的教育工作者既要培养学科理论研究和基本系统的开发人才，还要培养应用系统开发人才，甚至是应用人才。从层次上来讲，则需要培养系统的设计、实现、使用与维护等各个层次的人才。这就要求我国的计算机教育按照定位的需要，从知识、能力、素质三个方面进行人才培养。

硕士研究生的教育须突出“研究”，要加强理论基础的教育和科研能力的训练，使学生能够站在一定的高度去分析研究问题、解决问题。硕士研究生要通过课程的学习，进一步提高理论水平，为今后的研究和发展打下坚实的基础；通过相应的研究及学位论文撰写工作来接受全面的科研训练，了解科学研究的艰辛和科研工作者的奉献精神，培养良好的科研作风，锻炼攻关能力，养成协作精神。

高素质创新型计算机人才应具有较强的实践能力，教学与科研相结合是培养实践能力的有效途径。高水平人才的培养是通过被培养者的高水平学术成果来反映的，而高水平的学术成果主要来源于大量高水平的科研。高水平的科研还为教学活动提供了最先进的高新技术平台和创造性的工作环境，使学生得以接触最先进的计算机理论、技术和环境。高水平的科研也为高水平人才的素质教育提供了良好的物质基础。

为提高高等院校的教学质量，教育部最近实施了精品课程建设工程。由于教材是提高教学质量的关键，必须加快教材建设的步伐。为适应学科的快速发展和培养方案的需要，要采取多种措施鼓励从事前沿研究的学者参与教材的编写和更新，在教材中反映学科前沿的研究成果与发展趋势，以高水平的科研促进教材建设。同时应适当引进国外先进的原版教材，确保所有教学环节充分反映计算机学科与产业的前沿研究水平，并与未来的发展趋势相协调。

中国计算机学会教育专业委员会在清华大学出版社的大力支持下，进行了计算机科学与技术学科硕士研究生培养的系统研究。在此基础上组织来自多所全国重点大学的计算机专家和教授们编写和出版了本系列教材。作者们以自己多年来丰富的教学和科研经验为基础，认真研究和结合我国计算机科学与技术学科硕士研究生教育的特点，力图使本系列教材对我国计算机科学与技术学科硕士研究生的教学方法和教学内容的改革起引导作用。本系列教材的系统性和理论性强，学术水平高，反映科技新发展，具有合适的深度和广度。同时本系列教材两种语种（中文、英文）并存，三种版权（本版、外版、合作出版）形式并存，这在系列教材的出版上走出了一条新路。

相信本系列教材的出版，能够对提高我国计算机硕士研究生教材的整体水平，进而对我国大学的计算机科学与技术硕士研究生教育以及培养高素质创新型计算机人才产生积极的促进作用。

陈火旺

前 言

计算机网络是21世纪社会数字化、网络化与信息化的基础。与电力系统、通信系统一样，计算机网络已经成为支持现代社会整体运行的基础设施，人们须臾不能离开。但是，我们必须清醒地认识到：计算机网络是一把高悬在全人类头上的双刃剑。人类社会对计算机网络的依赖程度越高，网络安全就显得越重要。网络安全是网络技术研究中的一个永恒主题。

网络安全是一个充满活力与机遇的领域。网络安全技术研究与教育要注意以下10个方面的问题：

(1) 人类创造了网络虚拟社会的繁荣，也制造了网络虚拟社会的问题。网络安全是现实社会问题在网络虚拟社会中的反映。

(2) 当前网络的3大公害是：网络攻击、网络病毒与垃圾邮件。网络威胁的趋利性特征已经凸现，网络犯罪的专业化与黑色产业链正在逐步形成。

(3) 在“攻击—防御—新攻击—新防御”的循环中，网络攻击技术与网络反攻击技术相互影响、相互制约，共同发展、演变和进化。目前网络攻击的目的已经从最初好奇、玩世不恭与显示技艺高超，发展到经济利益驱动的有组织犯罪，甚至是恐怖活动。同时，网络攻击已经延伸到政治与军事等领域。

(4) 正如现实世界危害人类健康的各种“病毒”，像SARS与甲型H1N1流感病毒一样，它只会随着时间在演变，不可能灭绝。只要人类存在，就一定会存在危害人类健康的病毒。同样，只要网络存在，计算机与网络病毒就一定会存在。网络是传播计算机病毒的重要渠道。计算机病毒也会伴随着计算机与网络技术的发展而演变，不可能停止和灭绝。病毒是计算机与网络永远的痛。

(5) 随着互联网用户数量的剧增，网络广告的经济效应日益显现，在经济利益驱动下，垃圾邮件正呈现日趋严重的态势，已经造成了巨大的经济损失与社会问题。当前，网络攻击、病毒与垃圾邮件呈现出相互渗透、相互利用的趋势。如何检测、遏制网络攻击、病毒与垃圾邮件的蔓延已经成为网络安全最重要的研究课题。

(6) 密码学是网络安全研究的一个重要的工具，但是它并不能解决所有的问题。密码学涉及的是数字、公式与逻辑。数学是完美的，而现实社会却无法用数学准确描述。数学是精确和遵循逻辑规律的，而计算机和网络安全涉及的是人，人与人之间的关系以及人和机器之间的关系。人是有欲望的，是不稳定的，甚至是难于理解的。网络安全性存在于计算机硬件与软件、网络以及人的身上。

(7) 网络安全是一个系统的社会工程。网络安全的研究涉及技术、文化、道德与法制环境等多个方面。网络安全性是一个链条，它的可靠程度取决于链条中最薄弱的环节。同时

实现网络安全性是一个过程，不是任何一个产品可以替代的。在加强网络安全技术研究的同时，必须加快网络法制的建设，加强人们网络法制观念与道德的教育。

(8) 网络安全问题已经上升到国家安全的战略地位。由于计算机网络与互联网已经应用于现代社会的政治、经济、文化、教育、科学研究与社会生活的各个领域，因此说"发达国家和大部分发展中国家都是运行在网络之上"，这已经不会让人们感到吃惊了。社会生活越依赖于网络，网络安全必然会成为影响社会稳定、国家安全的重要因素之一。我国政府高度重视网络安全技术的研发与政策法规的制定。

(9) 自主研发网络安全技术、发展网络安全产业，建立自主可控的信息安全体系是关系到社会稳定与国家安全的重大问题。每个国家必须立足于本国，研究网络安全技术，培养专门人才，发展网络安全产业，才能构筑本国的网络与信息安全防范体系。哪个国家不高度重视网络与信息安全的技术研究与人才培养，必将在未来的国际竞争中处于被动和危险的境地。

(10) 支撑和服务我国信息社会、信息产业的网络安全产品与服务的核心技术必须由我国的技术专家掌握。这是事关国家安全、社会稳定、产业健康发展的重要保障因素。

从事网络安全与信息安全专业的技术人员可以分为工程师、高级工程师与专家等多个层次，社会对各个层次人才的需求都非常强烈。大学教育如果只能培养使用网络安全产品的人才是远远不够的。现在我国网络安全产品的研发很多是在国外公开的网络安全开源软件上进行改进。这种方法从表面上看是一条"捷径"，能够"立竿见影"，可以很快见"成效"，但是我们必须清醒地认识到这是一种"短视"的行为，并且存在巨大和潜在的危机，对于要真正形成具有我国自主知识产权的网络安全产业是非常不利的。

创新是一个民族的灵魂，而在网络与信息安全领域培养具有创新能力的高水平人才，产生创新性研究成果，开发具有自主知识产权的产品尤为重要。作为是多年从事网络安全技术研究与教育的教师和科研工作者，大家深知自己的责任重大，同时也深刻地认识到当前我国网络安全课程的教学水平还远不能满足国家与社会的要求。在网络安全教学过程中，教学内容与当前技术的发展水平、理论教学与实际工作能力的培养差距明显。这些问题将严重地制约网络安全人才的培养质量与学生的就业竞争力，从长远发展看也将严重地制约具有自主知识产权的技术与产品的发展。大学在对高层次信息安全专门人才的培养上必须要正视这个问题，要下苦功夫从基础开始，培养能够产生创新思想与具备研发能力的专门人才。

本书具有如下几个特点：

(1) 作者队伍是由南开大学信息技术科学学院计算机系、国家计算机病毒应急处理中心的人员组成的，具有多年信息安全科研工作、我国计算机病毒应急处理工作，以及本科、硕士与博士研究生教学的实践经验。作者通过总结多年来信息安全科研工作、计算机病毒应急处理工作以及本科、硕士与博士研究生教学工作实践经验，参考国内外知名信息安全技术研究部门与企业相关资料、文献的基础上，构思了全书的写作思路，设计了12个"近似实战"的网络安全软件设计与编程训练的课题。

(2) 网络安全软件编程训练课题可以分为：密码学及应用、网络安全常规技术、当前研究热点课题的综合训练等3个部分。训练课题覆盖了从密码学在网络通信中的应用，网络端口扫描、网络嗅探器、网络诱骗、网络入侵检测、安全Web、防火墙，到Linux内核网络协

议栈程序加固、网络病毒与垃圾邮件的检测与防治技术。训练课题接近学科研究的前沿，覆盖了网络安全研究的主要方向。

(3) 完成网络安全训练课题的操作系统环境选择为 Linux，完成训练课题不需要限定特殊的硬件环境和编程语言。这样做的目的是希望能够充分利用 Linux 开源软件的优势，通过在 Linux 环境中完成网络安全软件的设计与编程训练，提高读者研发具有自主知识产权的技术与产品的能力。

(4) 从研究生和高级人才培养的角度，应该强调"研究型"与"自主型"的学习方式。对于研究生教学过程来说，学生应该变被动的"听课、做笔记"转向主动、研究地学习和提高。从任课教师与导师角度应该强调"因材施教"。不同基础和不同需求的读者可以根据个人的基础、学习与工作的需要，选读其中的某些章节，完成其中部分课题的编程任务。

与本书作为姊妹篇的是《计算机网络高级软件编程技术》。这两本软件编程训练教材的写作风格是统一的，只是训练的目的和重点不同。《计算机网络高级软件编程技术》训练的目的是帮助读者掌握设计与开发网络应用系统的能力；《网络安全高级软件编程技术》训练的目的是帮助读者掌握设计与开发网络安全系统的能力。《计算机网络高级教程》与《计算机网络高级软件编程技术》、《网络安全高级软件编程技术》构成了一套计算机及相关专业研究生关于网络理论研究、网络应用系统与网络安全系统设计、编程能力培养的系列教材。

研究生教材不应该仅是一本一学期使用的教科书，更应该是一本技术参考书，甚至是一本手册。导师可以根据需要选择教材中的部分内容，作为基本的学习要求。学生学习的过程应该在导师的指导下有选择地自学和阅读，完成编程训练。有些内容可能第一次仅仅是读过和了解，如果今后科研、开发工作需要，可以再回过头来继续阅读和参考。

作为研究生与网络安全高级人才能力培养的教材，希望读者能够在阅读书中相关章节之后，独立地完成课题的编程任务。从严格训练的角度出发，书中只提供了解决问题的思路，给出了启发读者的编程示例，书后所附的光盘中给出了编程所需要的编程工具与测试数据集，希望读者通过阅读相关的章节，结合自己已经掌握的网络知识与基本编程方法，独立地完成训练课题的要求，克服浮燥的情绪，摒弃抄袭的陋习，通过下苦功夫、扎扎实实地训练，深入理解理论知识，提高实践能力，使研究生与从事网络安全工作的工程技术人员在学习过程中体会到"研究型"与"自主型"学习的快乐。

全书由吴功宜构思、组织编写与统稿。第 1 章由吴功宜执笔完成，第 2 章由陈志执笔完成，第 4 章、第 9 章和第 12 章由董大凡执笔完成，第 5 章、第 8 章、第 10 章由许昱玮执笔完成，第 3 章、第 7 章由王雪飞执笔完成，第 6 章、第 11 章由胡紫执笔完成，第 13 章由张建忠执笔完成，第 14 章由张健、杜振华和舒心执笔完成。

在本书的写作过程中得到了我国著名的信息安全专家沈昌祥院士的多方指导，得到了南开大学信息技术科学学院计算机系"网络与信息安全研究室"的老师和同学们的很多帮助，特别感谢刘瑞挺教授、徐敬东教授以及苏明副教授、吴英副教授、古力老师的帮助。

同时，本书也是在与国家计算机应急处理中心合作建设计算机病毒防治技术国家工程实验室等项目中所取得的成果。在写作过程中得到了国家计算机病毒应急处理中心张津弟主任的悉心指导，在此表示诚挚的感谢，同时感谢国家计算机病毒应急处理中心全体同志的支持与帮助。

本书可以作为计算机、信息安全、软件工程、通信工程、电子信息及相关专业的硕士与工

程硕士研究生、博士研究生的教材以及本科计算机专业、信息安全专业高年级网络安全课程的教材和参考书，也可作为网络安全软件编程高级人才的培训教材与研发工作参考手册。

限于作者的学术水平，书中错误与不妥之处在所难免。我们衷心地希望读者的批评指正，共同提高我国网络安全教学、研究与产业研发水平。

作者
于南开大学信息技术科学学院网络与信息安全研究室
国家计算机病毒应急处理中心
2009 年 10 月 10 日

目录

"十一五"国家重点图书　计算机科学与技术学科前沿丛书
计算机科学与技术学科研究生系列教材

第 1 章 网络安全课程内容、编程训练要求与教学指导

本章在系统总结网络安全技术特点、网络安全课程包括的主要内容的基础上，对网络安全软件编程训练课题设置的目的与训练要求，以及相关的教学方法及进度安排提出建议。

1.1 网络安全技术的特点

1.1.1 网络安全与现代社会安全的关系

生活在现实世界的人类创造了网络虚拟社会的繁荣，同时也造成了网络虚拟社会的问题。现实世界中真善美的东西，网络的虚拟社会都有。同样，现实社会中丑陋的东西，网络的虚拟社会一般也会有，只是表现形式不一样。如果透过复杂的技术术语和计算机屏幕，人们会发现：计算机网络的虚拟社会和现实社会之间，在很多方面都存在着“对应”关系。现实社会中人与人在交往中形成了复杂的社会与经济关系，在网络社会中，这些社会与经济关系以数字化的方式延续着。图 1-1 形象地描述了这个规律。

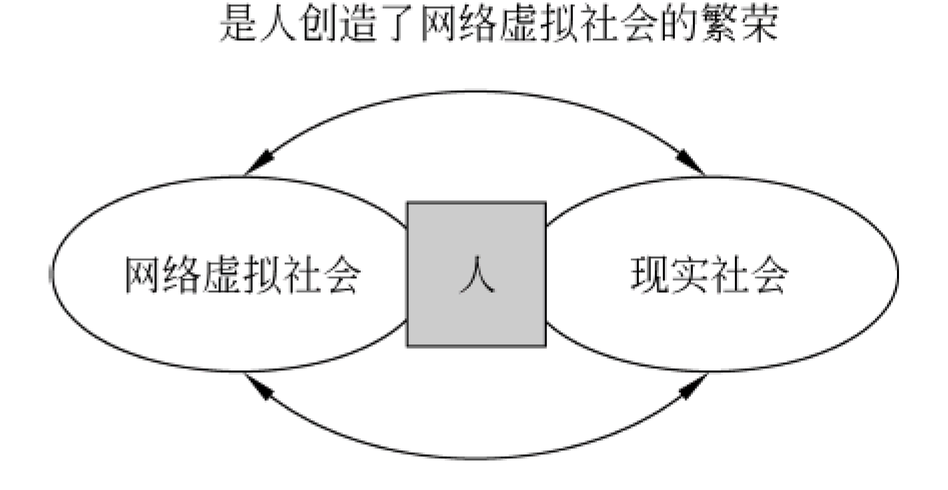


图 1-1　网络虚拟社会与现实社会的关系示意图

网络安全是现实社会安全的反映。网络安全问题实际上是个社会问题，光靠技术来解决这些问题是不可能的。网络安全是一个系统的社会工程，它涉及技术、政策、道德与法律法规等多方面。

1.1.2 网络安全与信息安全的关系

应用是网络存在和发展的理由。所有的信息系统与现代服务业都是建立在计算机网络与 Internet 环境之中的。正是由于这个原因，可以说网络应用系统的安全都是建立在计算机网络安全的基础之上的。图 1-2 给出了网络安全与计算机安全、应用系统安全以及用户信息安全关系的示意图。

用户的各种信息被保存在不同类型的应用系统之中，这些应用系统都是建立在不同的计算机系统之中的。计算机系统包括硬件、操作系统、数据库系统等，它们是保证各类信息

系统正常运行的基础。而运行信息系统的大型服务器或服务器集群及用户的个人计算机都是以固定或移动的方式接入到计算机网络与Internet中的。任何一种网络功能的服务实现都需要通过网络在不同的计算机系统之间多次进行数据与协议信息交换。例如,每一个人都会使用银行卡购物,使用电子邮件发送和接收邮件,利用浏览器访问Web站点,使用QQ与朋友聊天。

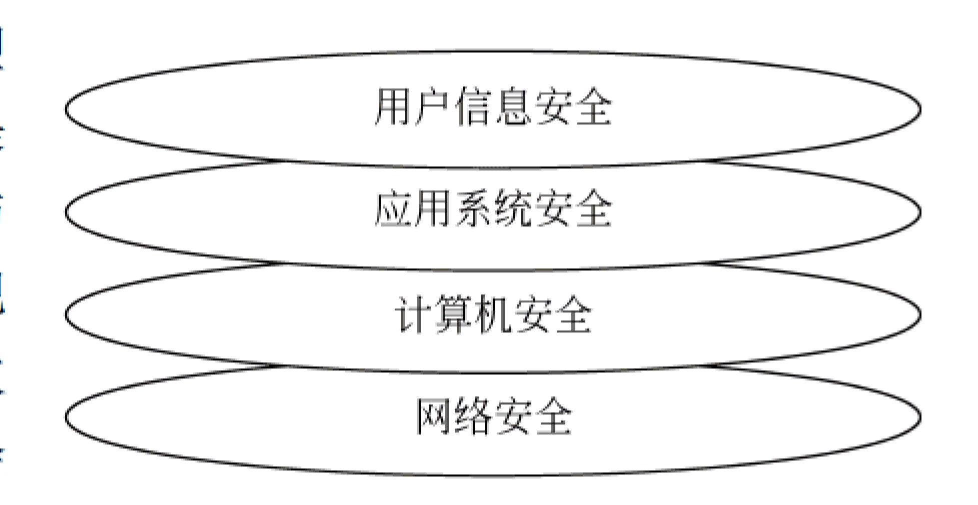


图1-2 网络安全与计算机安全、应用系统安全、用户信息安全关系示意图

病毒、木马、蠕虫、脚本攻击代码等恶意代码利用E-mail、FTP与Web系统进行传播,网络攻击、网络诱骗、信息窃取也都是在网络环境中进行的。网络安全是信息系统安全的基础,不能保证网络的安全性,信息系统的安全性就无从谈起。因此,网络安全研究是信息安全研究的重要组成部分,也是信息安全研究的基础。

1.1.3 网络安全与网络新技术的关系

按照正常人的思维方式,一位技术人员在研究和开发一种基于网络的新的应用技术与系统时,只会想到这种应用可以给人们的生活和工作带来什么样的好处和乐趣,一般不会想到黑客或居心不良的人会利用这种技术做什么坏事。而黑客恰恰是一类逆向思维和不按正常规律办事的人,他们不遵守正常人所遵循的道德规范。“Everything over IP,IP over everything”说明了计算机网络技术的成功,但是它所带来的问题也是网络技术人员始料未及的。P2P是一种十分有价值的网络应用模式,但是P2P除了可以方便信息共享之外,同时也给恶意代码的传播提供了一种新的途径。手机病毒的出现与无线射频标识RFID芯片可能感染病毒的研究结果公布,表明移动设备将成为黑客和恶意软件编写者下一个主攻的目标。

网络技术不是在真空之中,计算机网络是要提供给全世界的用户使用的,网络技术人员在研究和开发一种新的基于网络的应用技术与系统时,必须面对这样一个复杂的局面。成功的网络应用技术与成功的应用系统的标志是功能性与安全性的统一。网络安全问题不应该简单地认为是从事网络安全技术工程师的事,也是每位信息技术领域的工程师与管理人员需要共同面对的问题。

1.1.4 网络安全与密码学的关系

密码学是信息安全研究的重要工具,密码学在网络安全中有很多重要的应用,但是网络安全涵盖的问题远远超出了密码学涉及的范围。人们对密码学与网络安全的关系的认识有一个过程,这个问题可以用美国著名的密码学专家Bruce Schneier在《Secrets and Lies: Digital Security in a Networked World》一书的前言中讲述的观点来说明。Schneier在1996年出版了一本在信息安全方面非常经典的书《Applied Cryptography》。2000年他又出版了该书的第二版。他在第二版的前言中说明,他写第二版的动机之一是为了纠正第一版的一个错误。他说,在第一版中“我描述了一个数学的乌托邦:密码算法能将你最深的秘密保持数千年”。但是,他现在认为:“事实并非如此,密码学并不能做那么多的事。”密码学并非存

在于真空之中。密码学是数学的一个分支,它涉及数字、公式与逻辑。数学是完美的,而现实社会却无法用数学准确地描述。数学是精确的和遵循逻辑规律的,而计算机和网络安全涉及的是人所知道的事,人与人之间的关系以及人和机器之间的关系。人是有欲望的,是不稳定的,甚至是难于理解的。Schneier 在出版该书的第一版之后就成了美国分析和设计一些大型信息系统安全问题的顾问。但是后来的经历告诉他,安全性的弱点与数学“毫无关系”,它们存在于硬件、软件、网络和人的身上。他认识到:“安全性是一个链条,它的可靠程度取决于链条中最薄弱的环节。”同时,他认为:“安全性是一个过程,而不是一种产品。”

从一位数学家的认识转变中得到的启示是:密码学是研究网络安全所必需的一个重要的工具与方法,但是网络安全研究所涉及的问题要广泛得多。

1.1.5　网络安全与国家安全战略的关系

1. 网络安全对国家安全的影响

由于计算机网络与互联网已经应用于现代社会的政治、经济、文化、教育、科学研究与社会生活的各个领域,因此说“发达国家和大部分发展中国家都是运行在网络之上的”,这已经不会让人们感到吃惊了。社会生活越依赖于网络,网络安全就必然会成为影响社会稳定、国家安全的重要因素之一。下面引用了一些人的话从另外一个角度去印证这个观点。

2000 年 1 月 7 日美国前总统克林顿签署的《美国国家信息系统保护计划》中有这样一段话:“在不到一代人的时间内,信息革命和计算机在社会所有方面的应用,已经改变了我们的经济运行方式,改变了我们维护国家安全的思维,也改变了我们日常生活的结构”。

著名的未来学家托尔勒预言:“谁掌握了信息,谁控制了网络,谁就将拥有世界。”著名的军事预测学者在《下一场世界战争》一书中预言:“在未来的战争中,计算机本身就是武器,前线无处不在,夺取作战空间控制权的不是炮弹和子弹,而是计算机网络里流动的比特和字节。”

在“攻击—防御—新攻击—新防御”的循环中,网络攻击技术与网络安全技术在一起演变和发展,这个过程不会停止。但是,网络安全问题似乎已经超出了技术和传统意义上的计算机犯罪的范畴,开始发展成为一种政治与军事手段。在 2002 年,James F. Dunnigan 出版的《黑客的战争——下一个战争地带》一书中描述,2001 年阿富汗战争中美国为了配合武装战争,实施了信息战、新闻战和黑掉对方银行账户等各种信息战方法。2001 年 11 月 11 日以来,有记录显示很多中东与其他地区的黑客试图黑掉美国发电厂的网站,并试图进入美国和欧洲的核电站控制系统。电力控制、通信管理、城市交通控制、航空管制系统、GPS 系统与大型楼宇智能控制系统都建立在计算机网络之上,今后都可能成为黑客和网络战(cyberwar)攻击的目标。

“cyber”一词来源于希腊语,指的是“控制”。这些以前只出现在电影大片中的故事,已经变成现实必须预防和解决的问题。一场成功的网络战争的关键是计划和网络系统弱点。计划包括人力、技术、工具与网络武器等条件的准备。弱点的大小则取决于对方对网络的依赖程度,以及对网络系统安全建设的重视程度。

2. 信息时代国家安全战略重点的转移

美国和一些发达国家都已经将防范和应对攻击与破坏关键信息基础设施作为信息时代国家安全战略的重点。这个问题可以从美国最近举行的大规模“网络风暴”演习中清楚地看出。

2006 年美国、英国等 4 国举行了“网络风暴Ⅰ”演习。演习模拟了恐怖分子、黑客等发起破坏性网络攻击,导致能源、运输和医疗系统瘫痪,网络银行和销售系统出错,软件公司发售光盘染毒等危险时的应急处置措施与能力。

2008 年 3 月 10～14 日美国国土安全部举行了第 2 次为期一周、代号为“网络风暴Ⅱ”的网络战争演习,以全面检验国家网络安全和应急能力。这次演习耗资 620 万美元,仅制订预案就耗时 18 个月。参加本次演习的共有 18 个联邦机构、9 个州、40 家公司及 5 个国家。此次演习采用模拟真实环境中对网络的攻击情况,演习项目包括黑客入侵、网络欺诈、对服务器攻击等 1800 个项目。演习模拟了政府机构、银行、通信、信息、能源、航空与铁路运输等重要行业的网络系统遭受联合攻击时,网络安全专家应对攻击的处理能力。演习的目的就是针对各部门、各企业的网络漏洞,检验它们的网络应急计划和遭袭击后迅速恢复的能力,检验国家网络安全状况和应急处理协调能力。演习还设置了一些环节,检验美国政府绝密的“爱因斯坦计划”网络安全软件对入侵检测和网络系统安全的监控能力。“爱因斯坦计划”是研究和开发用于检测美国政府网站是否遭到入侵,以及监控整个网络系统安全的软件系统。

美国国土安全部的官员说,国外机构和私营企业参与演习是非常必要的。因为私营高科技信息公司控制了美国 80%的信息产业设施,它们在美国信息安全中扮演着重要角色。同时一次针对美国的网络攻击可能是从世界上不同的地方发起的,并且这些攻击可能是针对或利用私营企业拥有的网络基础设施进行的。

这次演习的参与者还要接受辨别真假威胁的挑战。协助制订演习预案的国土安全部的官员说,演习参与者需要通过电子邮件、电话及模拟电视新闻频道,了解他们面对的问题和网络现状,但其中有一些是混淆视听的假信息。这些干扰信息用于分散情报分析人员和公司负责人员的注意力。本项测试的目的是检验他们鉴别信息真伪的能力。

3. 发达国家互联网应用新的动向

美国在网络安全方面一直不遗余力。2009 年 5 月,美国总统奥巴马批准公布了国家网络安全评估报告,指出来自网络空间的威胁已成为美国面临的最严重的经济和军事威胁之一。6 月,美国国防部长罗伯特·盖茨正式宣布成立“网络战”司令部,成为世界上第一个创建网络战司令部的国家。9 月,国外媒体报道说,美国参议院提交了一份议案,计划赋予总统在紧急状态下关闭互联网服务的权利,但因遭到本国互联网企业和个体的强烈反对而搁置。10 月 1 日,美国国土安全部长珍妮特·纳波利塔诺表示,她所领导的部门已经获准在未来 3 年招募 1000 名网络安全专家。这是美国继设立网络战司令部之后,在网络安全领域采取的又一重要措施。同时,美国还要求全社会的个人和机构都参与到网络安保中来,从而将网络安保工作渗透到了全社会。报道援引纳波利塔诺的话说“这项新的招募权将允许国土安全部招募顶尖的网络分析师、开发人员和工程师,通过领导国家反网络威胁的工作为国家服务”。纳波利塔诺表示,新招募的网络专家将“帮助国土安全部完成保卫国家网络基础设施、系统和网络的广泛任务”。“新招募的网络专家将完成各种任务,如网络风险和战略分

析、网络事故回应、漏洞监控与评估、情报与调查，还有网络和系统工程”。美国总统奥巴马宣布 2009 年 10 月为美国“国家网络安全意识月”，并在发言中说：“网络攻击及其感染网络、设备和软件的病毒能力必须引起全美国人的关注。”他还表示，政府有责任将“数字基础设施作为一项战略性的国家资产”，“是国家安全的优先重点”。

从这些动态中，我们可以清醒地认识到：网络安全问题已成为信息化社会的一个焦点。每个国家只有立足于本国，研究网络安全技术，培养专门人才，发展网络安全产业，才能构筑本国的网络与信息安全防范体系。自主研发网络安全技术，发展网络安全产业是关系到一个国家国计民生与国家安全的重大问题。哪个国家不重视网络与信息安全，它们必将在未来的国际竞争中处于被动和危险的境地。

1.2　网络安全形势的演变

1.2.1　Internet 安全威胁的总体发展趋势

计算机网络与 Internet 是高悬在全人类头上的一把双刃剑。一个方面，计算机网络与 Internet 的应用对于各国的政治、经济、科学、文化、教育与产业的发展起到了重要的推动作用。另一方面，人们也对它的负面影响忧心忡忡。因此，网络安全的研究一直是伴随着网络技术与应用的发展而进步的。网络安全是网络技术研究中的一个永恒的主题。

近年来 Internet 安全威胁的总体趋势是：

(1) 受经济利益驱动，网络攻击的动机已经从初期的恶作剧、显示能力、寻求刺激，逐步向有组织的犯罪方向发展，甚至是形成有组织的跨国经济犯罪。

(2) 网络罪犯正逐步形成黑色产业链，网络攻击日趋专业化和商业化。

(3) 网络犯罪活动的范围将随着 Internet 的普及，逐渐从经济发达国家与地区向一些发展中国家和地区发展。

(4) 网络攻击开始出现超出传统意义上的网络犯罪的概念，正在逐渐演变成某些国家或利益集团的重要的政治、军事工具，以及恐怖分子的活动工具。

图 1-3 给出了 2002—2009 年网络安全威胁特点的演变情况。从图中可以清楚地看出，网络攻击已经从早期的恶作剧和简单的破坏性攻击发展到有组织的犯罪，这一点必须引起我们的高度重视。

1.2.2　近期网络安全威胁的主要特点

网络安全威胁呈现出以家庭用户作为最主要的被攻击目标，网络 Web 浏览器仍然是攻击的主要目标，攻击活动的动机已经从恶作剧、寻求刺激、表现技术的高超，转向有组织的经济犯罪。近期网络安全威胁的主要特点如下。

1. 地下交易体系呈现专业化与商业化的趋势

地下交易体系呈现专业化与商业化的趋势。通过社交网站与假冒安全软件进行诈骗用户敏感信息的事件明显增加。在网络攻击中，90%的攻击是针对用户的机密信息，如银行账户、信用卡信息等，这种攻击目前仍呈增长趋势。

2002年

- 病毒成为网络安全的主要威胁；
- 网络攻击加剧，目标主要是高科技企业，以及金融服务与电力能源企业；

2003年

- 漏洞数量急剧上升；
- 混合威胁危险增大；

2004年

- 利用漏洞制造病毒时间缩短；
- 僵尸网络数量增加；
- 攻击主要针对电子商务与Web应用；
- 病毒开始向移动设备转移；

2005年

- 利用漏洞与僵尸网络、网页仿冒攻击加剧；
- 网络攻击从破坏逐渐转向以牟利为目的的经济犯罪；

2006年

- 僵尸网络攻击急剧增加；
- 以窃取用户机密信息为目的的攻击锁定金融服务与家庭用户；
- 网络攻击转向经济犯罪形势明朗；

2007年

- 网络攻击变化为多层攻击和网络钓鱼；
- 网络地下交易服务器明显增多；
- 网络攻击呈专业化趋势，逐渐形成黑色产业链；
- 网络经济犯罪有从发达国家向欠发达国家或地区转移的迹象；

2008年

- 网络攻击转向窃取用户机密信息；
- Web页成为窃取用户信息的主要渠道；
- 地下交易体系趋向专业化与商业化，交易活跃；
- 垃圾邮件数量继续攀升，90%通过僵尸网络传播；

2009年

- 网络攻击呈专业化趋势，初步形成黑色产业链；
- 智能手机、苹果MAC与iPhone成为新的攻击目标；
- 网络病毒、垃圾邮件继续呈上升趋势；
- 利用假冒安全软件诱骗、窃取敏感信息的攻击呈上升趋势；

图 1-3　网络安全威胁特点的演变示意图

2. 智能手机成为新的攻击目标

3G 手机的应用，使得能够访问互联网的智能手机成为网络攻击的下一个目标，病毒制作者已经瞄准苹果机的 MAC 操作系统与 iPhone 软件制作新病毒。第一款针对 iPhone 软件的病毒已经开始流传。

3. 受网络病毒感染的网页数量与垃圾邮件数量持续攀升

网页已取代网络成为攻击活动的主要渠道，越来越多的在线用户会因为访问一些日常的网站而受到感染。根据统计，2009 年上半年每 3.6 秒钟就有一个网页被病毒感染，其感染网页的数量是 2008 年同期的 4 倍。2009 年的商业邮件中的 89.7%是垃圾邮件。

4. Web 2.0 与搜索引擎服务成为黑客攻击的重点

Web 2.0 与搜索引擎开放的网络应用程序接口 API，使得它们又一次成为黑客攻击和利用的目标。随着越来越多的网络服务的推出，黑客利用 Web API 服务来获得用户信任，达到窃取用户信息的目的。

1.3 网络安全技术研究的基本内容

1.3.1 网络安全技术研究内容的分类

网络安全技术研究的目的是保证网络环境中传输、存储与处理信息的安全性。总结近年来网络安全研究的内容、方法与技术的发展，可以将网络安全研究归纳为如图 1-4 所示的结构。

网络安全技术研究主要包括以下 4 个方面的内容。

1. 网络安全体系结构的研究

网络安全体系结构的研究主要涉及网络安全威胁分析、网络安全模型与确定网络安全体系以及对系统安全评估的标准和方法的研究。根据对网络安全威胁的分析，确定需要保护的网络资源，对资源攻击者、攻击目的与手段、造成的后果进行分析；提出网络安全模型，并根据层次型的网络安全模型，提出网络安全解决方案。网络安全体系结构研究的另一个重要内容是系统安全评估的标准和方法，这是评价一个实际网络应用系统安全状况的标准，是提出网络安全措施的依据。

2. 网络安全防护技术

网络安全防护技术的研究涉及防火墙技术、入侵检测技术与防攻击技术、防病毒技术、安全审计与计算机取证技术以及业务持续性技术。

3. 密码应用技术

密码应用技术的研究涉及包括对称密码体制与公钥密码体制的密码体系，以及在此基

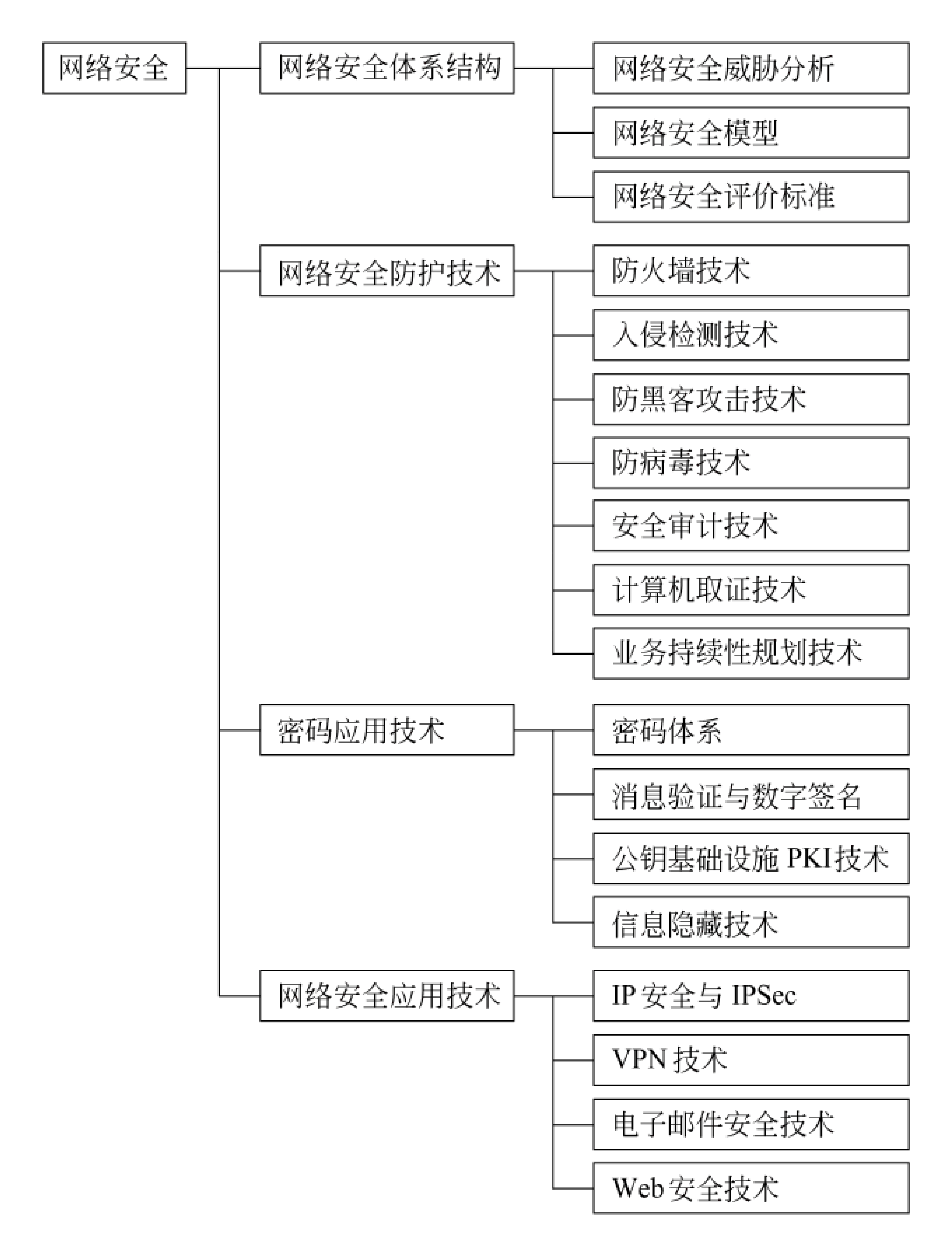


图 1-4　网络安全技术研究内容的分类图

础上主要研究的消息认证与数字签名技术、信息隐藏技术、公钥基础设施 PKI 技术。

4. 网络安全应用

网络安全应用技术研究主要包括 IP 安全、VPN 技术、电子邮件安全、Web 安全与网络信息过滤技术。

1.3.2　网络攻击的分类

1. 网络攻击的基本概念

有经验的网络安全人员都有一个共识：知道自己被攻击就赢了一半。但是问题的关键是：怎么知道自己已经被攻击了。入侵检测技术就是检测入侵行为，因此研究入侵检测技术是网络安全研究中最重要的课题之一。

在十几年之前，网络攻击还仅限于破解口令和利用操作系统漏洞的有限的几种方法，然而随着网络应用规模的扩大和技术的发展，在 Internet 上黑客站点随处可见，黑客工具可以任意下载，黑客攻击活动日益猖獗。黑客攻击已经对网络的安全构成了极大的威胁。研究黑客攻击技术，了解并掌握攻击技术，才有可能有针对性地进行防范。研究网络攻击方法已

经成为制定网络安全策略、研究入侵检测技术的基础。法律对攻击的定义是指：攻击仅仅发生在入侵行为完全完成，并且入侵者已在目标网络内。但是对于网络安全管理员来说，一切可能使网络系统受到破坏的行为都应视为攻击。

目前网络攻击大致可以分为：系统入侵类攻击、缓冲区溢出攻击、欺骗类攻击与拒绝服务 DoS 攻击等。

系统入侵类攻击者的最终目的都是为了获得主机系统的控制权，从而破坏主机和网络系统。这类攻击又分为信息收集攻击、口令攻击和漏洞攻击。缓冲区溢出攻击是指：通过往程序的缓冲区写超出其限制长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其他的指令。缓冲区攻击的目的是使攻击者获得程序的控制权。网络欺骗的主要类型有：IP 欺骗、ARP 欺骗、DNS 欺骗、Web 欺骗、电子邮件欺骗、源路由欺骗、地址欺骗与口令欺骗等。

2. 网络安全威胁的层次

网络安全威胁可以分为 3 个层次：主干网络的威胁、TCP/IP 协议安全威胁与网络应用的威胁。

主干网络的威胁主要表现在主干路由器与 DNS 服务器。攻击主干网最直接的方法是攻击主干路由器与 DNS 服务器。全球 13 台根域 DNS 服务器支持着整个 Internet 的运行。1997 年 7 月的人为错误曾经导致根域 DNS 服务器工作不正常，致使 Internet 系统局部服务中断。2002 年 8 月黑客利用 Internet 主干网的 ASN No.1 信令存在的安全漏洞，攻击了主干路由器、交换机和一些基础设施，造成了严重的后果。2002 年 10 月 21 日美国东部时间下午 4：45 开始，13 台根域 DNS 服务器遭受了规模最大的分布式拒绝服务 DDoS (Distributed Denial of Service)攻击，导致其中的 9 台根域 DNS 服务器工作不正常。

3. 服务攻击与非服务攻击的基本概念

Internet 中的网络防攻击可以归纳为以下两种基本类型：服务攻击与非服务攻击。

服务攻击(application dependent attack)是指对为网络提供某种服务的服务器发起攻击，造成该网络的"拒绝服务"，使网络工作不正常。特定的网络服务包括 E-mail、Telnet、FTP、Web 服务等。

非服务攻击(application independent attack)不针对某项具体应用服务，而是针对网络层及低层协议进行的。攻击者可能使用各种方法对网络通信设备(例如路由器、交换机)发起攻击，使得网络通信设备工作严重阻塞或瘫痪。

4. 网络攻击手段的分类

网络攻击手段的分类如图 1-5 所示。

网络攻击手段很多并且不断地变化，总结目前出现的主要的网络攻击现象与手段大致可以将它们分为：欺骗类攻击、DoS/DDoS 类攻击、信息收集类攻击、漏洞类攻击等 4 种基本类型。

(1) 欺骗类攻击

欺骗类攻击的手段主要包括：口令欺骗、IP 地址欺骗、ARP 欺骗、DNS 欺骗与源路由

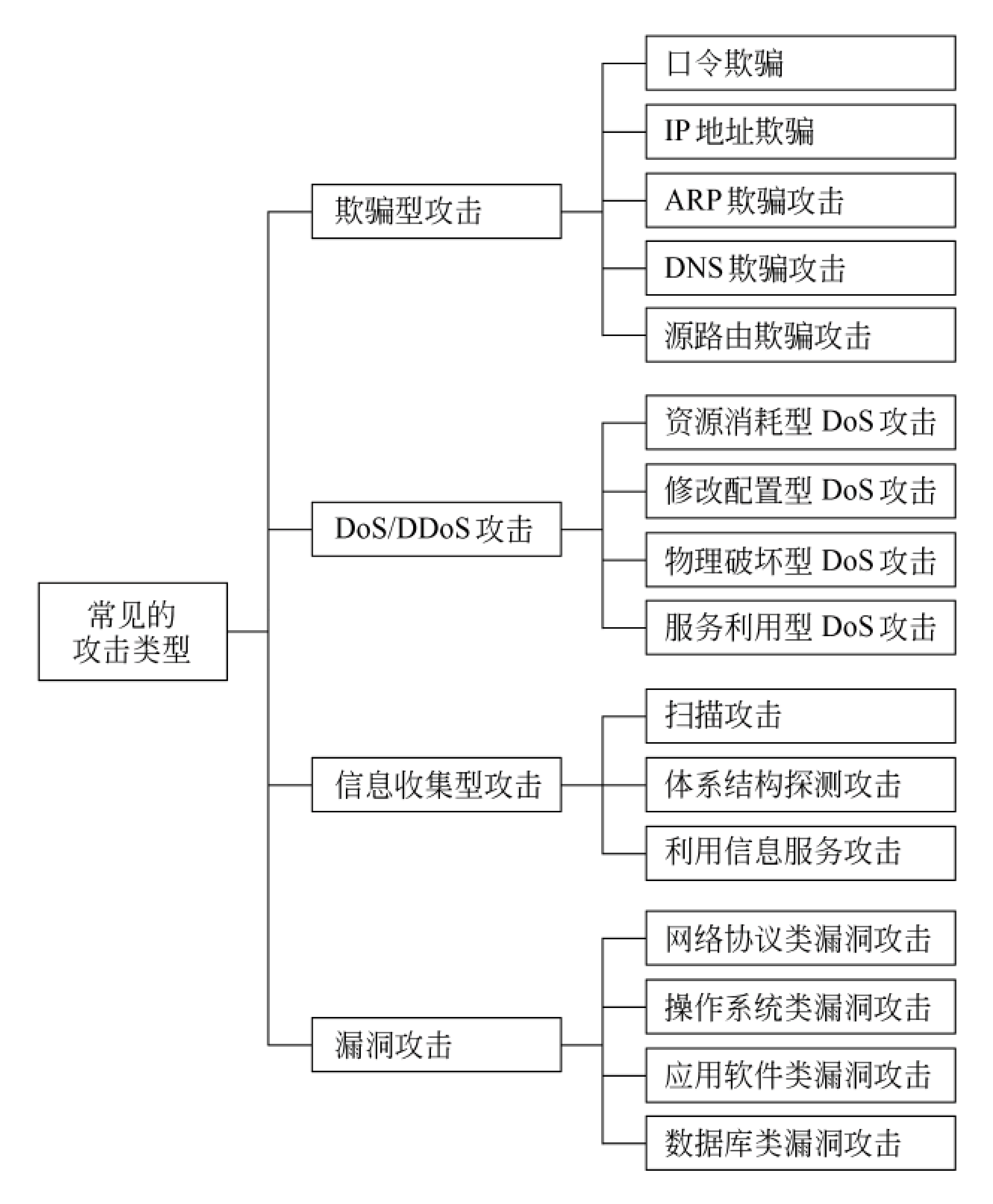


图 1-5 网络攻击手段的分类图

欺骗等。

(2) DoS/DDoS 攻击

拒绝服务 DoS 攻击与分布式拒绝服务 DDoS 攻击的手段主要包括：资源消耗型、修改配置型、物理破坏型与服务利用型等类型的拒绝服务攻击。

(3) 信息收集类攻击

信息收集类攻击的手段主要包括：扫描攻击、体系结构探测攻击和利用服务攻击等。

(4) 漏洞类攻击

漏洞类攻击的手段主要包括：网络协议类、操作系统类、应用软件类与数据库类等。

同时需要注意的是，网络安全漏洞实际上分为：技术漏洞与管理漏洞两大类，这里主要考虑的是技术漏洞类的问题。

5. 典型的网络攻击：DoS 攻击与 DDoS 攻击

(1) 拒绝服务攻击

拒绝服务(Denial of Service，DoS)攻击主要是通过消耗网络系统有限的、不可恢复的资源，从而使合法用户应该获得的服务质量下降或遭到拒绝。DoS 攻击最本质的是延长正常网络应用服务的等待时间，或者使合法用户的服务请求遭到拒绝。DoS 攻击的目的不是闯入一个站点或者是更改数据，而是使站点无法服务于合法的服务请求。

拒绝服务 DoS 攻击大致可以分为 4 类：资源消耗型 DoS 攻击、修改配置型 DoS 攻击、

物理破坏型 DoS 攻击和服务利用型 DoS 攻击。

① 资源消耗型 DoS 攻击

资源消耗型 DoS 攻击通过消耗网络带宽、内存和磁盘空间、CPU 利用率，使网络系统不能正常工作。常见的方法是：攻击者制造大量广播包或传输大量文件，占用网络链路与路由器带宽资源；攻击者制造大量电子邮件、错误日志信息、垃圾邮件等，占用主机中共享的磁盘资源；攻击者制造大量的无用信息或进程通信交互信息，占用 CPU 和系统内存资源。

② 修改配置型 DoS 攻击

修改配置型 DoS 攻击通过修改系统运行配置，阻止合法用户的使用和网络的正常工作。常见的方法是：改变路由信息；修改 Windows NT 的注册表；修改 UNIX 的各种配置文件。

③ 物理破坏型 DoS 攻击

物理破坏型 DoS 攻击通过破坏网络、计算机或系统物理支持环境，使网络系统不能正常工作。常见的方法是：破坏计算机系统；破坏路由器和通信线路；破坏网络与计算机设备供电或机房空调系统。

④ 服务利用型 DoS 攻击

服务利用型 DoS 攻击利用网络或协议的漏洞达到攻击的目的。常见的方法如 Land 攻击、Ping to Death 攻击、TCP 标志位攻击、IP 碎片（teardrop）攻击、ICMP&UDP 洪泛攻击等。

(2) 分布式拒绝服务攻击

分布式拒绝服务（Distributed Denial of Service，DDoS）攻击是在 DoS 攻击基础上产生的一类攻击形式。DDoS 攻击采用了一种比较特殊的体系结构，攻击者利用多台分布在不同位置的攻击代理主机，同时攻击一个目标，从而导致被攻击者的系统瘫痪。

1.3.3 网络安全防护技术研究

1. 防火墙的基本概念

防火墙（firewall）是在网络之间执行控制策略的系统，它包括硬件和软件。在设计防火墙时，人们做了一个假设：防火墙保护的内部网络是“可信赖的网络”（trusted network），而外部网络是“不可信赖的网络”（untrusted network）。设置防火墙的目的是保护内部网络资源不被外部非授权用户使用，防止内部受到外部非法用户的攻击。因此防火墙安装的位置是在内部网络与外部网络之间。防火墙的主要功能包括：检查所有从外部网络进入内部网络的数据包；检查所有从内部网络流出到外部网络的数据包；执行安全策略，限制所有不符合安全策略要求的分组通过；具有防攻击能力，保证自身的安全性。防火墙通过检查所有进出内部网络的数据包，检查数据包的合法性，判断是否会对网络安全构成威胁，为内部网络建立安全边界（security perimeter）。

构成防火墙系统的两个基本部件是包过滤路由器（packet filtering router）和应用级网关（application gateway）。最简单的防火墙可以由一个包过滤路由器组成，而复杂的防火墙系统由包过滤路由器和应用级网关组合而成。由于组合方式有多种，因此防火墙系统的结构也有多种形式。包过滤技术基于路由器技术。包过滤路由器按照系统内部设置的分组过滤规则（即访问控制表），检查每个分组的源 IP 地址、目的 IP 地址，决定该分组是否应该转

发。普通的路由器只对分组的网络层报头进行处理,对传输层报头不进行处理,而包过滤路由器需要检查 TCP 报头的端口号字节。包过滤规则一般是基于部分或全部报头的内容。对于 TCP 报头信息,它可以是:源 IP 地址、目的 IP 地址、协议类型、IP 选项、源 TCP 端口号、目的 TCP 端口号与 TCP ACK 标识。实现包过滤的关键是制定包过滤规则。包过滤路由器将分析接收到的包,按照每一条包过滤的规则加以判断,凡是符合包转发规则的包被转发,凡是不符合包转发规则的包则被丢弃。包过滤路由器也叫屏蔽路由器。包过滤路由器是被保护的内部网络与外部不信任网络之间的第一道防线。

包过滤只能在网络层、传输层对进出内部网络的数据包进行监控,但是网络用户对网络资源和服务的访问发生在应用层,因此必须在应用层上建立用户身份认证和访问操作的检查和过滤功能,这个功能由应用级网关完成。应用代理是应用级网关的另一种形式,但是它们的工作方式不同。应用级网关以存储转发方式,检查和确定网络服务请求的用户身份是否合法,决定是转发还是丢弃该服务请求。因此应用级网关在应用层"转发"合法的应用请求。应用代理完全接管了用户与服务器的访问,隔离了用户主机与被访问服务器之间的数据包的交换通道。在实际应用中,应用代理的功能由代理服务器(proxy server)实现。应用级网关与应用代理的优点是可以针对某一特定的网络服务,并能在应用层协议的基础上分析与转发服务请求与响应。同时它们一般都具有日志记录功能。

防火墙是一个由软件与硬件组成的系统。不同内部网的安全策略与防护目的不同,防火墙系统的配置与实现方式也有很大区别。实际的防火墙系统经常是由包过滤路由器与应用级网关作为基本单元,采用多级的结构和多种组态。

2. 入侵检测技术研究的基本概念

(1) 入侵检测系统的基本功能

入侵检测系统(intrusion detection system,IDS)是对计算机和网络资源的恶意使用行为进行识别的系统。它的目的是监测和发现可能存在的攻击行为,包括来自系统外部的入侵行为和来自内部用户的非授权行为,并采取相应的防护手段。

1980 年 James Anderson 在"Computer Security Threat Monitoring and Surveillance"的论文中提出了入侵检测系统的概念。他对"网络入侵"的定义是:潜在的、有预谋的、未经授权的服务操作,目的是使网络系统不可靠或无法使用。

1987 年 Domthy Donning 在"An Intrusion Detection Model"的论文中提出入侵检测系统 IDS 的框架结构。入侵检测系统的功能主要有:监控、分析用户和系统的行为;检查系统的配置和漏洞;评估重要的系统和数据文件的完整性;对异常行为的统计分析,识别攻击类型,并向网络管理人员报警;对操作系统进行审计、跟踪管理,识别违反授权的用户活动。

IDS 作为一种主动式、动态的防御技术迅速发展起来,成为当前安全研究中一个新的热点。IDS 通过动态探查网络内的异常情况,及时发出警报,有效弥补了其他静态防御技术的不足。IDS 正在成为对抗网络攻击的关键技术,研究的总体目标是:智能、分布式、实时的网络入侵防御技术与系统。

(2) 入侵检测系统的分类

入侵检测信息的来源主要有:基于主机的信息源、基于网络的信息源、应用程序日志信息与入侵检测系统的报警信息。入侵检测系统 IDS 的分类如图 1-6 所示。

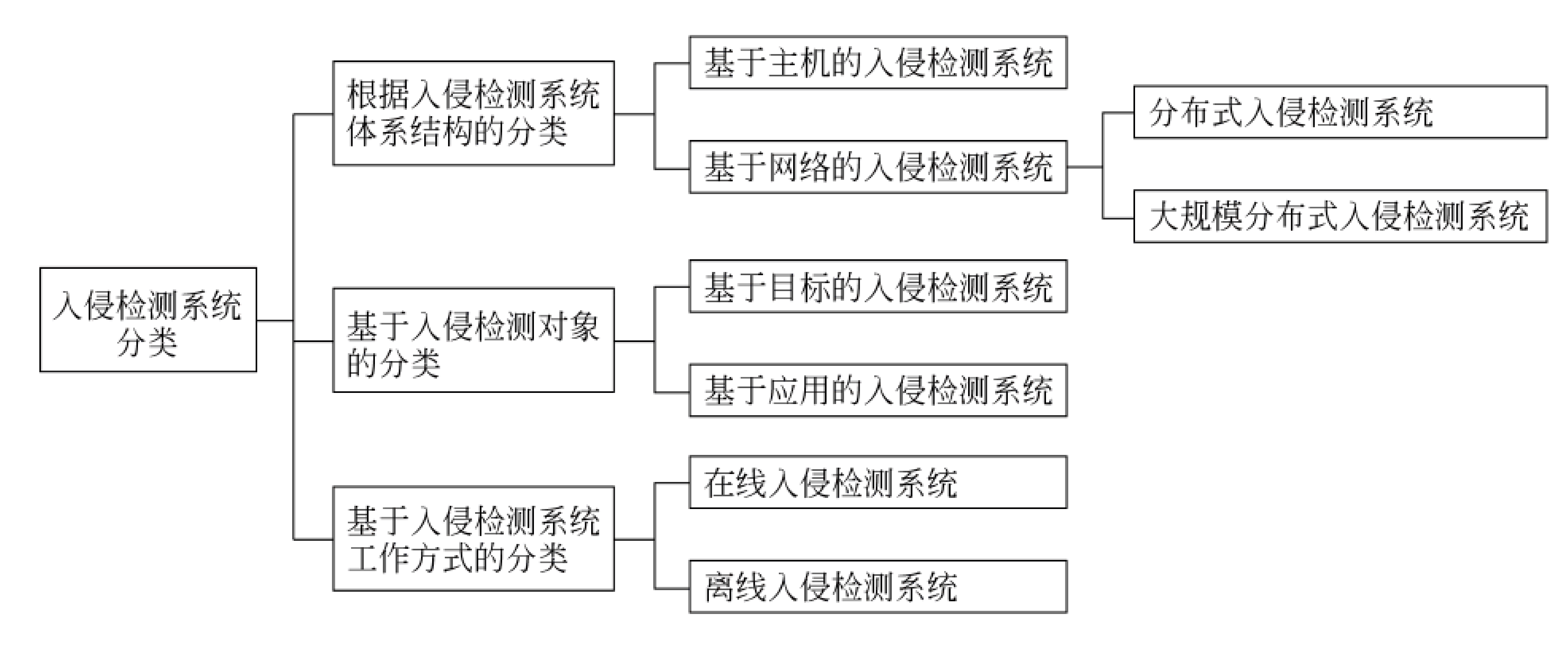


图 1-6　入侵检测系统 IDS 的基本分类图

根据入侵检测的体系结构不同，入侵检测系统 IDS 可以分为基于主机的入侵检测系统 IDS 与基于网络的入侵检测系统 IDS 等两大类。基于主机的入侵检测系统是一种集中式的 IDS，基于网络的入侵检测系统是一种分布式的 IDS。

目前的网络环境要求入侵检测系统能够根据不同子网所发生的入侵迹象，判断整个网络可能出现的分布式入侵的现象与程度，因此研究多个子网的入侵检测系统的协同工作的大规模分布式入侵检测系统是下一阶段的一个重要课题。

根据检测的对象和基本方法的不同，入侵检测系统 IDS 又可以分为：基于目标的入侵检测系统和基于应用的入侵检测系统。

根据入侵检测的工作方式不同又可分为离线检测系统和在线检测系统。离线检测系统是非实时工作的系统。它是事后分析审计事件，从中检测入侵活动。在线检测系统是实时联机的测试系统，它包含实时网络数据包的审计分析。

3. 安全审计的基本概念

安全审计是一个安全的网络必须支持的一个功能，它是对用户使用网络和计算机所有活动记录分析、审查和发现问题的重要手段。安全审计对于系统安全状态的评价，分析攻击源、攻击类型与攻击危害，收集网络犯罪证据至关重要。

国际上已经在 TCSEC(Trusted Computer System Evaluation Criteria)中对信息系统的安全等级与评价方法发布了标准，并提出了对安全审计的基本要求。TCSEC 对 Accountability 提出的要求是：审计信息必须被有选择地保留和保护，与安全有关的活动能够被追溯到负责方，系统应能够选择和记录与安全有关的重要信息，以便将审计的开销减少到最小，提高安全审计的有效性。

在 C2 等级安全中标准也对审计有确定的要求：系统能够创建和维护审计数据，保证审计数据记录不被删除、修改和非法访问。

1998 年 ISO 与 IEC 公布的《信息技术安全评估通用准则》(2.0 版)的 11 项安全功能需求中，明确规定了网络安全审计的功能：安全审计自动响应、安全审计事件生成、安全审计分析、安全审计预览、安全审计事件存储、安全审计事件选择等。因此，一个网络系统是否具备完善的审计功能是评价系统安全性的重要标准之一。

安全审计研究的内容主要有：网络设备及防火墙日志审计、操作系统日志审计。目前防火墙等安全设备具有一定的日志功能，在一般情况下只记录自身的运转情况与简单的违规操作信息。由于一般的网络设备与防火墙对网络流量分析能力不够强，所以这些信息还不能对网络的安全提供分析依据。同时，由于一般网络设备与防火墙采用内存记录日志，因此空间有限，信息需要经常地覆盖。因此没有能力提供足够的分析数据。这种网络设备与防火墙的设计不能满足安全评测标准的要求。

目前大多数操作系统都提供日志功能，记录用户登录等信息，但是如果要从大量零散的信息去人工分析安全信息是很困难的。同时，日志被修改的可能性也存在。因此，目前多数操作系统安全审计方法尚不能满足安全评测标准的要求。

1.3.4 网络防病毒技术研究

恶意传播代码(malicious mobile code，MMC)是一种软件程序，它被设计成能够从一台计算机传播到另一台计算机，从一个网络传播到另一个网络，目的是在网络和系统管理员不知情的情况下，对系统进行故意地修改。恶意传播代码包括病毒、木马、蠕虫、脚本攻击代码，以及垃圾邮件、流氓软件与恶意的 Internet 代码。

病毒程序的名称来源类似于生物学的病毒。病毒程序是一种专门修改其他宿主文件或硬盘的引导区，来复制自己的恶意程序。一旦感染病毒，宿主文件就变成病毒再去感染其他的文件。

木马程序又叫做特洛伊木马，是一种非自身复制程序。它伪装成一种程序，但是程序是什么用户并不知道。例如，用户从网络上下载并运行了一个游戏程序，但游戏程序的制造者同时将一个木马程序装进了用户的计算机，以便黑客进入并控制该计算机。木马程序不改变或感染其他的文件。后门(backdoor)程序是恶意程序中的子程序，它使黑客可以访问本来安全的计算机系统，而不会让用户或管理员知道。

蠕虫是一种复杂的自身复制代码，它完全依靠自身进行传播。蠕虫典型的传播方式是利用广泛使用的应用程序，如电子邮件、聊天室等。蠕虫可以将自己附在一封要被发送的邮件上，或者在两个互相信任的系统之间，通过一条简单的 FTP 命令来传播。蠕虫一般不寄生在其他文件或引导区中。

蠕虫与木马有很多共同点，它们之间的主要区别在于：木马总是假扮成其他的程序，而蠕虫是在后台暗中破坏；木马依靠用户的信任去激活它，而蠕虫从一个系统传播到另一个系统，不需要用户的介入；木马不对自身进行复制，而蠕虫对自身进行大量复制。

根据病毒的传染性可以分为：引导型病毒、文件型病毒、复合型病毒。

根据病毒的连接方式可以分为：源码型病毒、入侵型病毒、操作系统型病毒。

根据病毒的破坏性可以分为：良性病毒、恶性病毒。

网络病毒问题的解决，只能从采用先进的防病毒技术与制定严格的用户使用网络的管理制度两个方面入手。

1.3.5 计算机取证技术研究

1. 计算机取证的作用

计算机取证(computer forensics)在网络安全中属于主动防御技术，它是应用计算机辨

析方法，对计算机犯罪的行为进行分析，以确定罪犯与犯罪的电子证据，并以此为重要依据提起诉讼。针对网络入侵与犯罪，计算机取证技术是一个对受侵犯的计算机与网络设备与系统进行扫描与破解，对入侵的过程进行重构，完成有法律效力的电子证据的获取、保存、分析、出示的全过程，是保护网络系统的重要技术手段。

计算机取证也叫计算机法医学。对于构建对入侵者有威慑力的主动网络防御体系有重要的意义，是一个极具挑战性的研究课题。在信息窃取、金融诈骗、病毒与网络攻击日益严重的情况下，计算机取证技术与相关法律、法规的研究、制定已经迫在眉睫。

2. 电子证据的概念

计算机取证的主要任务是获取电子证据。证据是法官判定犯罪嫌疑人是否有罪的标准。电子证据 20 世纪 30 年代首先出现在美国，而我国在法庭出示电子证据还是近十年的事。1976 年美国出台了有关电子证据的法律条款，我国在这方面正在加快相关法律法规的研究和制定。

国际组织 IOCE 对于电子证据有相关的定义。他们认为证据可以分为电子证据、原始电子证据、电子证据副本与拷贝等 4 类。电子证据是指在法庭上可能成为证据的二进制形式存储或传送的信息。原始电子证据是指在查封计算机犯罪现场获取的相关物理介质、存储的数字信息。电子证据副本是指从原始电子证据获取的所有数字信息的完全拷贝。

与传统意义上的证据一样，电子证据必须是可信的、准确的、完整的且符合法律法规的，必须能够被法庭接受的。电子证据有它自己的特点：表现形式的多样性、准确性和易修改性。电子证据可以存储在计算机的硬盘、软盘、内存、光盘与磁带中，可以是文本、图形、图像、语音、视频等各种形式的。如果没有人蓄意破坏，那么电子证据是准确的，能够反映事件的过程与某些细节。电子证据不受人的感情与经验等主观因素的影响。但是，电子证据是非常容易被修改的。

3. 计算机取证方法

计算机取证的方法基本可以分为：静态方法和动态方法。

传统的取证是在案发之后或已经造成严重后果之后对现场的取证，这种取证属于静态取证。静态取证由于缺乏实时性和连续性，因此在法庭上缺乏说服力，有时会因为作案者已经销毁了证据，而无法起诉。同时，由于是事后处理，因此即使作案者受到了法律制裁，但是危害已经造成了。

动态取证方法通过实时监控攻击的发生，在启动响应系统，判断危害的严重程度，作相应处理的同时，对入侵过程实施同步取证和详细记录。网络攻击一般要经过嗅探、入侵、破坏、掩盖入侵足迹等过程，在攻击的每个过程中，网络安全系统都需要采取 IDS 系统与“蜜罐”技术相结合的方法完成取证。

1.3.6 网络业务持续性规划技术研究

1. 网络业务持续性规划技术的基本概念

网络文件备份恢复属于日常网络与信息系统维护的范畴，而业务持续性规划涉及对突

发事件对网络与信息系统影响的预防技术。

在实际的网络运行环境中,数据备份与恢复功能非常重要。数据一旦丢失,可能会给用户造成不可挽回的损失。网络数据可以进行归档与备份。归档与备份是有区别的。归档是指在一种特殊介质上进行永久性存储,归档的数据可能包括文件服务器不再需要的数据,但是由于某种原因需要保存若干年,一般将这些数据存放在一个非常安全的地方。网络数据备份是一项基本的网络维护工作。对于网络管理员来说,网络文件备份是日常的网络管理工作任务之一。网络文件备份要解决以下几个基本问题:选择备份设备、选择备份程序与建立备份制度。

在20世纪网络技术处于发展过程中,人们的注意力主要集中在“建设”上。进入21世纪,网络已经广泛地应用于社会生活的各个方面,人们对广域网与城域网的运行提出了“电信级”与“准电信级”运营的要求,银行、电信、社会保险、政府的网络与数据的安全已经成为影响社会稳定的因素,因此网络系统的安全性,以及对于突发事件的应对能力已经被提到重要的位置。

对于网络与信息系统的业务持续性规划技术,Jon William Toigo的著作《Disaster Recovery Planning Preparing for the Unthinkable(Third Edition)》已经做了最好的诠释。Jon William Toigo作为一位美国航运公司网络与计算机系统的负责人,亲身经历了9·11事件。他在书中写道:“除了9·11事件所产生的社会与政治后果,这场灾难很特别的一点,或许是发现如此众多的受到影响的机构,竟然没有任何灾难恢复规划。对于世贸中心的440多家商业机构,曼哈顿区被电力、通信和设备中断所影响的成千上万家公司,还有五角大楼中众多的政府机构,只有一小部分——或许是200家——有预先制订的灾难恢复规划”。在这本书中,Jon William Toigo讨论了一个术语的深层次含义,那就是“灾难”与“业务持续性”的区别。一些专家认为“灾难”一般是指洪水、飓风与地震之类的自然灾害所造成的危害。而在信息技术领域中指的是:由于网络基础设施的中断所导致公司业务流程的非计划性中断。造成业务流程的非计划性中断的原因除了洪水、飓风与地震之类的自然灾害、恐怖活动之外,还有网络攻击、病毒与内部人员的破坏,以及其他不可抗拒的因素。这些突发事件的出现,其结果是造成网络与信息系统、硬件与软件的损坏,以及密钥系统与数据的丢失,关键业务流程的非计划性中断。针对各种可能发生的情况,必须针对可能出现的突发事件,提前做好预防突发事件出现造成重大后果的预案,控制突发事件对关键业务流程所造成的影响。因此,现在人们倾向于将“灾难恢复规划”的术语改为“业务持续性规划”。

2. 业务持续性规划的基本内容

业务持续性规划技术的研究大致包括以下一些基本内容:规划的方法学问题、风险分析方法和数据恢复规划。

我国的网络与网络应用技术正处于快速发展阶段,中大型网络与网络信息系统的数量正在高速增长,但是在企业、部门与公司主管层对网络系统的风险与业务持续性的认识还有待提高,适合我国经济与技术发展水平、认识能力的业务持续性技术的研究与应用还处于起步和示范工程阶段,研究适合我国国情的设计方法与应用系统开发是一项重要的任务。

1.3.7　密码学在网络中的应用研究

1. 密码体系的基本概念

密码技术是保证网络与信息安全的基础与核心技术之一。密码学(crytography)包括密码编码学与密码分析学。密码体制的设计是密码学研究的主要内容。人们利用加密算法和一个秘密的值(称为密钥)来对信息编码进行隐蔽,而密码分析学试图破译算法和密钥。两者相互对立,又互相促进地向前发展。

密码体制是指一个系统所采用的基本工作方式以及它的两个基本构成要素,即加密/解密算法和密钥。

加密的基本思想是伪装明文以隐藏它的真实内容,即将明文伪装成密文。伪装明文的操作称为加密,加密时所使用的信息变换规则称为加密算法。由密文恢复出原明文的过程称为解密。解密时所采用的信息变换规则称作解密算法。

加密算法和解密算法的操作通常是在一组密钥控制下进行的。加密密钥和解密密钥相同的密码体制称为对称密码(symmetric cryptography)体制。加密密钥和解密密钥不相同的密码体制称为公钥密码(asymmetric cryptography)体制。密钥可以看做是密码算法中的可变参数。改变了密钥,实际上也就改变了明文与密文之间等价的数学函数关系。密码算法是相对稳定的,而密钥则是一个变量。现代密码学的一个基本原则是:一切秘密寓于密钥之中。在设计加密系统时,加密算法是可以公开的,真正需要保密的是密钥。

对于同一种加密算法,密钥的位数越长,密钥空间(key space)越大,即密钥可能的范围越大,破译的困难就越大,安全性就越好。一种自然的倾向就是使用最长的可用密钥,它使得密钥很难被猜测出来。但是密钥越长,进行加密和解密过程所需要的计算时间也将越长。

公钥密码技术对信息的加密与解密使用不同的密钥,用来加密的密钥是可以公开的,而用来解密的密钥是需要保密的,因此又被称为公钥加密(public key encryption)技术。

1976 年 Diffie 与 Hellman 提出了公钥加密的思想,加密用的密钥与解密用的密钥不同,公开加密密钥不至于危及解密密钥的安全。用来加密的公钥(public key)与解密的私钥(private key)是数学相关的,并且公钥与私钥成对出现,但是不能通过公钥计算出私钥。公钥密钥密码体系在现代密码学中非常重要。按照一般的理解,加密主要是解决信息在传输过程中的保密性问题。但是还存在另一个问题,即如何对信息发送人与接收人的真实身份进行验证,防止对所发出信息和接收信息的用户在事后抵赖,并且能够保证数据的完整性。公钥密钥密码体制对这两个方面的问题都给出了很好的回答。公钥加密技术可以大大简化密钥的管理,如果网络中要对 N 个用户之间进行通信加密,只需要使用 N 对密钥就可以了。公钥加密技术与对称密钥加密技术相比,其优势在于不需要共享通用的密钥,用于解密的私钥不需要发往任何地方。公钥可以通过 Internet 进行传递与分发。公钥加密技术的主要缺点是加密算法复杂,加密与解密的速度比较慢。公钥加密技术常用于数据完整性、数据保密性、防抵赖与发送端身份认证等方面。

2. 消息验证与数字签名的研究

2004 年 8 月 28 日第十届全国人大常委会第十一次会议表决通过了《数字签名法》,于

2006年4月1日正式实施。《数字签名法》是我国首部"真正意义上的信息化立法"，它的重点是：确定数字签名的法律效力；规范电子签名的行为；明确认证机构的法律地位及认证程序；规定电子签名的安全保证措施。随着《数字签名法》的颁布实施，研究与应用电子签名技术就显得更加重要了。

在网络环境中，消息验证与数字签名是防止主动攻击的重要技术。消息验证与数字签名的主要目的是：验证信息的完整性，验证消息发送者身份的真实性。实现消息验证需要使用以下技术：消息的加密、消息验证码MAC与散列Hash函数。

利用数字签名可以实现以下功能：

(1) 保证信息传输过程中的完整性

安全单向哈希函数的特性保证如果两条信息的信息摘要相同，则它们的信息内容也相同。因此，可以通过比较发送前的信息摘要与接收到的信息摘要来判断信息在传输过程中是否被篡改或改变过。由于在传输过程中信息摘要是经过发送端的私钥加密的，其他人生成相同加密摘要的可能性很小，因此，如果重新计算的信息摘要与解密后的信息摘要相同，就可以证明该信息在传输过程中没有被篡改或改变过。

(2) 发送端的身份认证

数字签名技术使用公钥加密算法，发送端使用自己的私钥对发送的信息进行加密。接收端可以通过发送端的公钥解密接收到的信息。这样，接收端便可以证实该信息是由发送端发送的，同时也可以确认发送端的身份。因为除了发送端之外，没有人可以生成这样的密文。

(3) 防止交易中的抵赖发生

当交易中的抵赖行为发生时，接收端可以将接收到的密文呈现给第三方。由于该密文由发送端的私钥生成，其他任何人都不可能产生该密文，而发送端的公钥是对公众公开的，任何人都可以获得该公钥，任何人都可以解开该密文。这样，第三方就可以通过公钥解密接收方呈送的密文，同时判断发送方是否发生了抵赖行为。

目前，数字签名技术的研究主要集中在：不可否认签名、防失败签名、盲签名与群签名等方面。

3. 身份认证技术的研究

身份认证可以通过以下3种基本途径之一或它们的组合来实现。

(1) 所知(knowledge)：个人所掌握的密码、口令等。

(2) 所有(possesses)：个人的身份证、护照、信用卡、钥匙等。

(3) 个人特征(characteristics)：人的指纹、声纹、笔迹、手型、脸型、血型、视网膜、虹膜、DNA，以及个人动作方面的特征等。根据安全要求和用户可接受的程度，以及成本等因素，可以选择适当的组合，来设计一个自动身份认证系统。

网络环境中个人身份认证的新技术是下一阶段网络安全研究的重点问题之一。对安全性要求较高的系统，由口令和证件等提供的安全保障是不完善的。口令可能被泄露，证件可能丢失或被伪造。更高级的身份验证是根据用户的个人特征来进行确证，它是一种可信度高，而又难于伪造的验证方法。新的、广义的生物统计学方法正在成为网络环境中个人身份认证技术中的最简单而安全的方法。它是利用个人所特有的生理特征来设计的。个人特征

包括很多,如容貌、肤色、发质、身材、姿势、手印、指纹、脚印、唇印、颅相、口音、脚步声、体味、视网膜、血型、遗传因子、笔迹、习惯性签字、打字韵律,以及在外界刺激下的反应等。当然,采用哪种方式还要看是否能够方便地实现,以及是不是能够被用户所接受。

个人特征都具有“因人而异”和随身“携带”的特点,不会丢失且难于伪造,适用于高级别个人身份认证的要求。因此,将生物统计学与网络安全、身份认证结合起来是目前网络安全研究的一个重要课题。

4. 公钥基础设施 PKI 的研究

公钥基础设施(Public Key Infrastructure,PKI)是利用公钥加密和数字签名技术建立的提供安全服务的基础设施。它为用户建立一个安全的网络运行环境,使用户能够在多种应用环境之下方便地使用加密的数字签名技术,从而保证网络上数据的机密性、完整性与不可抵赖性。一个 PKI 系统对用户是透明的和安全的,用户在获得加密和数字签名服务的时候,不需要知道 PKI 是如何管理证书与密钥的。

PKI 的主要任务是确定可信任的数字身份,而这些身份可以用来与密码机制相结合,提供认证、授权或数字签名验证等服务。PKI 系统实现的关键是密钥的管理。因此一个实用的公钥基础设施 PKI 包括:认证中心(Certificate Authority,CA)、注册认证(Registration Authority,RA)中心、策略管理、密钥与证书的管理、密钥的备份与恢复等。数字证书是公钥密码体制中的权威电子文档,也是网络环境中的身份证,用来证明一个主体(用户与服务器等)身份与它使用的私钥的合法性的数字 ID。

作为 PKI 的一种应用,基于 PKI 的 VPN 研究也随着 B2B 电子商务的发展而发展。同时,基于 PKI 的应用还有很多,如 Web 服务器与浏览器之间的应用、安全电子邮件、电子数据交换、Internet 环境中的信用卡交易等。无线网络环境中的 PKI 系统的研究也是目前的重要课题。应该说,PKI 在一些发达国家已经进入了实际应用阶段,有了成熟的产品与应用系统。但是目前在我国仍然处于起步与示范工程阶段,随着电子商务、电子政务的发展,发展适合中国国情的 PKI 技术,研制和开发自主、完整和成熟的 PKI 产品非常迫切。

5. 信息隐藏技术的研究

信息隐藏(information hinding)也称为信息伪装。它是利用人类感觉器官对数字信号的感觉冗余,将一些秘密信息以伪装的方式隐藏在非秘密的信息之中,达到在网络环境中隐蔽通信和隐蔽标识的目的。信息加密是将明文变成第三方不认识的密文,第三方知道密文的存在,而信息隐藏技术是使第三方不知道密文的存在。信息隐藏技术由两个部分组成:信息嵌入算法、隐蔽信息检测与提取算法。如果从广义的信息隐藏技术的定义出发,目前信息隐藏技术研究的内容大致可以分为:隐蔽信道、隐写术、匿名通信与版权标志等 4 个方面。

1.3.8 网络安全应用技术研究

网络安全应用技术研究主要包括:IP 安全、VPN 技术、电子邮件安全、Web 安全。

1. IP 安全的研究

IP 协议本质上是不安全的,伪造一个 IP 分组,篡改 IP 分组的内容,窥探传输中分组的内容都是比较容易的。接收端不能保证一个 IP 分组确实来自于它的源 IP 地址,也不能保证 IP 分组在传输过程中没有泄露或被篡改。为了解决 IPv6 协议的安全性问题,IETF 于 1995 年成立了一个 IP 协议与密钥管理机制的组织,研究在 IP 协议上保证 Internet 数据传输安全性的标准。这个组织在几年的工作中,提出了一系列的协议,构成了一个安全体系(IP Security protocol,IPSec)。IPSec 具有以下几个特征:

(1) IPSec 是 IETF 在开发 IPv6 时为保证 IP 数据包安全而设计的,是 IPv6 协议的一部分。IPSec 可以向 IPv4 与 IPv6 提供互操作、高质量与基于密码的安全性。

(2) IPSec 提供的安全服务包括访问控制、完整性及数据原始认证等。这些服务在 Internet 的网络层提供,并向 Internet 的网络层与更高层提供保护。

(3) IPSec 协议实际上是一个协议包,而不是单一的一种协议。它的安全结构由 3 个主要的协议以及加密与认证算法组成,它包括认证头(authentication header,AH)协议和封装安全载荷(encapsulating security payload,ESP)协议,以及 Internet 安全关联密钥管理协议(internet security association and key management protocol,ISAKMP)、Internet 密钥交换(internet key exchange,IKE)协议。

IPSec 在 IP 层对数据分组进行高强度加密与验证服务,使得安全服务独立于应用程序,各种应用程序都可以共享 IP 层所提供的安全服务。

2. VPN 技术的研究

传统的大型企业网的组建方案中,为了实现处于不同地理位置的部门、分支机构 LAN 与 LAN 之间的远距离通信,除了租用 DDN 专线之外,没有其他的解决办法。在这种情况下,用户除了要承担昂贵的专线租金之外,还要承担繁杂地调制解调器与接入设备的管理任务,并且这种方法的扩展性与灵活性较差。随着帧中继 FR 网络与 ATM 网络的大量应用,使虚拟专网 VPN 技术成为可能。但是基于帧中继与 ATM 网络的 VPN 同样也存在着扩展性与灵活性差的缺点。不同电信运营商的帧中继与 ATM 网络互联存在着很大问题,VPN 的虚电路配置与维护要由运营商来完成,远距离的互联还涉及多个运营商之间的协调问题。在这样的背景下,基于 IP 的 VPN 技术表现出费用低、组网灵活与具有良好可扩展性的优点,因此受到电信业与网络界的普遍重视。随着 Internet 的大规模应用和移动办公人员数量的增加,1993 年欧洲虚拟专网联盟 EVUA 成立,并致力于 VPN 技术的研究与推广。VPN 是一种模拟“专用”广域网,通过构建安全网络平台,在公用通信网络的通信对端之间建立一条安全、稳定的通信隧道,为用户提供安全的通信服务。

VPN 通过隧道技术、密码技术、密钥管理技术、用户和设备认证技术来保证安全的通信服务,隧道技术是实现 VPN 功能的基本技术。VPN 的分类可以有多种方法,可以根据形成隧道的不同协议来进行分类,也可以根据 VPN 所使用的传输网络类型进行分类。

3. 电子邮件安全技术的研究

电子邮件存在的垃圾邮件、诈骗邮件、炸弹邮件、病毒邮件等问题已经引起了人们的高

度重视。未加密的数据在网络上很容易被截获，如果电子邮件不是数字签名的，那么用户无法确定邮件是从哪里发送来的。要解决电子邮件的安全问题，有3种研究途径：端到端的安全电子邮件技术、传输层的安全电子邮件技术以及邮件服务器安全技术。

4. Web安全技术的研究

Web是Internet中最重要的应用，因此Web安全受到威胁的因素也越多，这是很自然的，那么研究Web安全威胁就显得格外重要。来自网络的威胁和攻击很多，存在不同的分类。根据Web访问的结构分类，可以分为：对Web服务器的安全威胁、对Web浏览器的安全威胁和对通信信道的安全威胁等3类。

根据Web访问的攻击性质分类，可以分为：主动攻击与被动攻击等两类。

根据Web访问的威胁造成的后果分类，可以分为：对Web数据完整性的攻击、对Web数据保密性的攻击、对Web系统的拒绝服务攻击与对Web认证鉴别的攻击等4类。

对于攻击者来说，Web服务器、数据库服务器有很多弱点可以被利用，比较明显的弱点在服务器的CGI程序与一些工具程序上。Web服务的内容越丰富，应用程序越大，则包含错误代码的概率就越高。程序设计人员在编写CGI程序与一些工具程序时，一个简单的错误和不规范的编程都有可能为系统增加了一个安全漏洞。CGI程序和脚本程序可以驻留在任何部分，这种规定给软件编程带来了很多的方便，但同时也为攻击者提供了可乘之机。一个故意放置恶意代码的CGI程序能够自由地访问系统资源，使系统失效、删除文件、盗窃用户资料、控制服务器，而系统对CGI程序的跟踪和管理却很困难。

对于用户端的浏览器来说，静态页面由标准的HTML语言编制，其作用是显示页面内容和链接其他页面。但是随着动态网页技术的出现，情况发生了较大变化。动态页面是在静态页面之中嵌入对于用户是透明的应用程序，在用户浏览一个页面时，这些应用程序就会自动地被下载，并启动运行。企图破坏客户端的人就可以利用这个机制将具有破坏性的恶意程序自动下载到客户计算机中，窃取客户资料，控制用户计算机或删除用户信息。用户可以在Interent上下载的应用程序与工具程序非常多，用户无法判断哪些是恶意的。因此，用户所做的最危险的行为是：从网上任意下载程序，并在本机上运行。这在Web环境中几乎是每一位用户都有可能做的事。按照正常的规则，运行一个程序就相当于已经接受了程序开发人员的控制。对于开放的Web系统来说，面对大量的用户，安全的威胁随时都会出现。因此，研究应对Web安全的研究一直是一个富有挑战性的问题。从网络体系结构的角度，Web安全技术的研究可以分别从网络层协议、传输层协议和应用层协议着手。

Web安全涉及操作系统、数据库、防火墙、应用程序与其他多项安全技术，是一个系统的安全工程，不能简单地从Web协议的角度去解决问题。只有综合考虑各方面的因素，集成多种安全技术，才能够切实保证Web系统的安全。

1.4　网络安全技术领域自主培养人才的重要性

1.4.1　网络安全技术人才培养的迫切性

关于网络安全技术的重要性以及网络安全技术人才培养的迫切性问题，可以从美国政

府的两份文件的相关描述中得到重要的启发。

1991 年美国政府的咨询报告中有一段关于“计算机风险”的描述。他们写道：“我们处在风险之中。美国依赖计算机。计算机控制了供电、通信、航空和金融服务；它们被用来存储至关重要的信息，从医疗档案到企业计划，到犯罪记录。尽管我们信任计算机，但是仅就设计欠缺和缺乏足够的质量控制而言，它们具有易于受到事故和故意攻击的缺陷，这也是最值得人们警惕的。现代窃贼用计算机比用枪支能够偷到更多的东西。也许明天的恐怖分子能够用键盘比用炸弹造成更大的破坏。”

2000 年美国白宫的一份关于《保护信息系统国家计划》中写道：“重中之重是人员培训。你会看到，国家计划把信息空间的保护放在新的安全标准、多层次防御技术、新的科学研究，以及人员培训上。所有这些基础中，最紧迫、最困难，也是其他一切因素先决条件的，是一支受过训练的信息与计算机技术专家队伍。一个世纪前，当美国在迅速实现电力网络化的时候，它也迅速地为这一新的经济行业培训出电气工程师。现在，‘计划’提出措施来刺激高等教育的发展，以便培养美国在这一领域迫切需要的专门人才。”

2009 年我国 Internet 的网民人数已经达到 3.38 亿，居世界第一，各种网络应用快速发展。我们应该清醒地认识到，对于社会经济、科学与教育高速发展的中国来说，重中之重也是网络安全专业人员的培养。

1.4.2 网络安全技术人才培养的特点

从以上的讨论中可以看出以下几点关于网络安全技术的特点，以及社会对网络安全技术人才需求的发展趋势。

(1) 社会急需大量、各个层次的网络安全技术人才

社会对于网络的依赖程度越高，网络安全就越发显得重要。要保障涉及全社会各行各业与各个领域的网络与信息系统的安全，急需大量、各个层次的技术人才。随着网络应用的不断扩大和技术的日益发展，网络安全研究的内容将不断地变化和扩展，新的问题不断提出，各种新的网络安全软硬件产品需要研发，尤其是能够掌握网络安全系统设计与软件编程能力的高层次技术人才十分匮乏。

(2) 高层次网络安全人才培养是相当重要和艰巨的一项任务

网络安全涉及的内容非常广，不是简单地教学生如何选择网络安全设备，懂得安装调试网络安全设备就可以了。网络安全是一种应用性很强的技术，简单地通过上一门课，或者是读几本书是没有办法真正掌握这门技术的。网络安全技术涉及密码学、计算机软硬件技术、通信技术、微电子芯片设计技术、法律法规与网络行为学知识。一位好的高层次网络安全人才必须在以上几个方面具备很好的学术基础，具备很高的职业道德与法律意识。同时，设计一个好的网络安全产品，实际上是在同心术不正的黑客斗智斗勇。一个成功的网络安全技术人员必然要具有很强的事业心与研究精神。因此，培养高层次网络安全人才是相当困难的。同时，合格的高层次网络安全人才也必然是技术精英和社会的宝贵财富。大学研究生教育应当义不容辞地承担起培养合格的高层次网络安全人才的任务。

(3) 涉及我国网络安全的核心技术必须由我国技术人员掌握

由于网络安全关乎国家安全、社会稳定与民众的切身利益，因此涉及网络安全的核心技术必须立足于国内研究部门完成，关键网络安全设备的算法、软件、核心芯片都必须立足于

国内产业设计与生产。网络安全的核心技术，以及国家网络与信息安全关键管理岗位都必须由可靠的技术人员担任。这是各国都要遵循的原则。处于高速发展的中国政府、教育界与产业界对这个问题必须有清醒地认识。

随着计算机网络和 Internet 的应用越来越广泛，社会对网络依赖的程度也越来越高，社会对掌握网络安全知识与技能的高层次人才的需求必然会越来越强烈，这也就为大学毕业生提供了很多就业机会。因此，学习和掌握网络安全技术对于提高计算机及相关专业毕业生的就业竞争力也是非常有益的。

1.5　网络安全软件编程课题训练的基本内容与目的

1.5.1　基于 DES 加密的 TCP 聊天程序编程训练的基本内容与目的

DES 算法是一种典型的对称分组加密算法，也是应用密码学中最基本的加密算法之一，目前广泛应用于网络通信加密、数据存储加密、口令与访问控制系统之中。掌握 DES 算法在网络通信中的应用对于理解对称加密算法非常有益。这里以加密 TCP 聊天程序为任务，研究基于 DES 的通信加密应用软件的设计与编程方法。

训练的主要目的是：

(1) 理解对称加密算法 DES 的基本工作原理。

(2) 掌握将对称加密 DES 算法应用于网络通信的基本设计方法与实现技术。

(3) 掌握 Linux 操作系统 socket 编程的基本方法。

1.5.2　基于 RSA 算法自动分配密钥的加密聊天程序编程训练的基本内容与目的

在讨论了传统的对称加密算法 DES 原理与实现方法的基础上，这里将以典型的公钥密码体系中 RSA 算法为例，以基于 TCP 协议的聊天程序加密为任务，系统地讨论公钥密码体系 RSA 算法的基本工作原理与软件编程方法。

训练的主要目的是：

(1) 理解 RSA 算法的基本工作原理。

(2) 掌握将 RSA 算法应用于网络通信系统的基本设计方法与实现技术。

(3) 掌握在 Linux 操作系统中实现 RSA 算法的编程方法。

(4) 了解 Linux 操作系统异步 I/O 接口的基本工作原理。

1.5.3　基于 MD5 算法的文件完整性校验程序编程训练的基本内容与目的

MD5 算法是目前最流行的信息摘要算法，已广泛应用于数字签名、文件完整性检测等领域。熟悉 MD5 算法对于开发安全的网络应用程序具有重要意义。

训练的主要目的是：

(1) 理解 MD5 算法的基本原理。

(2) 掌握利用 MD5 算法生成数据摘要的计算方法。

(3) 掌握将 MD5 算法应用于文件完整性校验的基本设计与编程方法。

(4) 掌握在 Linux 操作系统中检测文件完整性的基本方法。

1.5.4 基于 Raw Socket 的 Sniffer 设计与编程训练的基本内容与目的

网络监控软件能够监控网络流量,发现网络中异常的数据流,了解黑客攻击的手段,有效地发现和防御网络攻击,是保证网络安全的重要工具和手段之一,也是网络安全技术人员必须掌握的重要技能之一。这里研究基于 Raw Socket 的 Sniffer 系统设计与软件编程方法。

训练的主要目的是:

(1) 理解网络嗅探器 Sniffer 的基本工作原理与实现方法。

(2) 掌握 Raw Socket 的基本工作原理。

(3) 掌握 TCP/IP、ICMP 协议原理及 socket 编程方法。

1.5.5 基于 OpenSSL 的安全 Web 服务器设计与编程训练的基本内容与目的

Web 使用的传输协议是 HTTP 协议。HTTP 采用明文传输,网络传输中的重要数据有被第三方截获的危险。安全超文本传输协议(Hypertext Transfer Protocol over SSL, HTTPS)用于保护敏感数据在 Web 系统中的传输安全。HTTPS 通过安全套接字协议层(Secure Socket Layer,SSL)加密 HTTP 数据,并可以与 HTTP 数据共存。因此,研究基于 OpenSSL 的安全 Web 服务器的设计与软件编程方法,对于提高 Web 系统的安全性有着重要的意义。

训练的主要目的是:

(1) 理解 HTTPS 协议与 SSL 协议的基本工作原理。

(2) 掌握使用 OpenSSL 编程的方法。

(3) 掌握安全 Web 系统设计的基本设计与编程方法。

1.5.6 网络端口扫描器设计与编程训练的基本内容与目的

网络端口扫描器是一种重要的网络安全检测设备,也是网络黑客攻击的重要工具之一。通过端口扫描,不仅可以发现目标主机的开放端口和操作系统的类型,还可以查找系统的安全漏洞,获得口令缺陷等相关信息。因此,掌握端口扫描的基本工作原理与软件设计方法是网络安全工程师必须掌握的基本技能之一。同时,研究网络端口扫描器的实现方法,对于维护网络系统的安全,了解黑客攻击的手段有着重要意义。

训练的主要目的是:

(1) 理解网络端口扫描器的基本结构、工作原理与设计方法。

(2) 掌握 TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描的基本工作原理、设计与实现方法。

(3) 掌握 ping 程序的设计与实现方法。

(4) 掌握 Linux 操作系统多线程编程的基本方法。

1.5.7　网络诱骗系统设计与编程训练的基本内容与目的

网络诱骗是一种主动网络防护与网络取证的主要手段之一，对于保护网络应用系统安全具有重要作用。这里在讨论网络诱骗基本原理的基础上，系统地研究系统结构设计与软件编程的基本方法。

训练的主要目的是：

（1）理解网络诱骗系统基本工作原理。

（2）理解 Linux 系统调用实现和原理，以及对系统调用的扩展方法。

（3）掌握 Loadable Kernel Module 编程的相关知识和方法。

（4）了解 Linux 系统中程序隐藏的方法。

1.5.8　入侵检测系统设计与编程训练的基本内容与目的

入侵检测系统（IDS）是一种对网络传输进行实时监视，并在发现可疑传输时发出警报或者采取主动防御措施的网络安全设备。入侵检测分为特征检测与异常检测。这里在系统分析入侵检测系统基本工作原理的基础上，以基于特征检测的入侵检测系统为任务，研究入侵检测系统的设计与软件编程方法。

训练的主要目的是：

（1）掌握基于特征的入侵检测系统的基本工作原理、设计与实现方法。

（2）掌握 K-Means 算法的计算过程。

（3）掌握在网络安全研究中应用数据挖掘技术的基本概念与方法。

1.5.9　基于 Netfilter 和 IPTables 防火墙系统设计与编程训练的基本内容与目的

网络上充斥着各种病毒、木马以及针对主机漏洞的攻击。如何使网络主机有效抵御各种非法入侵，保证重要数据的机密性和安全性已成为当前网络上一个亟待解决的问题。防火墙是保护网络资源的重要手段之一。这里以 Netfilter/IPTables 为工具，研究防火墙系统设计与软件编程的方法。

训练的主要目的是：

（1）理解防火墙技术的基本工作原理。

（2）理解 Linux 环境中 Netfilter/IPTables 的工作机制。

（3）掌握对 Netfilter 内核模块进行扩展编程的基本方法。

（4）掌握通过 IPTables 构建防火墙的基本方法。

1.5.10　Linux 内核网络协议栈加固编程训练的基本内容与目的

Linux 是一种开发源代码的操作系统，程序开发人员能够通过修改或升级其源代码对系统进行加固。这里通过加固 Linux 网络协议栈，改变 Linux 内核对孤立 TCP SYN 数据包的处理方式，研究提升系统对 TCP SYN 拒绝服务攻击的防御能力的基本方法。

训练的主要目的是：

（1）理解 TCP 连接建立过程，以及拒绝服务式攻击的基本原理与方法。

（2）结合 Linux 内核源代码的分析，理解 Linux 网络协议栈的实现原理。

(3) 掌握对 TCP SYN Flood 的防御手段,以及对 Linux 内核进行扩展开发的方法。

(4) 了解 Linux TCP cookie 防火墙的工作原理。

1.5.11 利用 Sendmail 收发和过滤邮件系统设计与编程训练的基本内容与目的

随着电子邮件应用的深入,垃圾邮件日趋泛滥。大量的垃圾邮件不仅增加了网络的负担,更为诈骗、病毒攻击等行为提供了方便。因此,掌握电子邮件相关的软件编程技术和基本的垃圾邮件过滤技术,对于软件人员至关重要。这里将利用 Sendmail 邮件服务器的 milter 接口,实现简单的垃圾邮件过滤功能,帮助读者掌握 Linux 环境下垃圾邮件过滤软件编程的基本思路和方法。

训练的主要目的是:

(1) 掌握 Linux 环境下利用 Sendmail 邮件服务器的 milter 接口的回调函数实现简单的垃圾邮件过滤软件编程的基本思路和方法。

(2) 掌握利用黑名单与白名单判断邮件的拒绝接收或转发软件的设计与编程方法。

(3) 掌握根据关键字过滤方法判断邮件的拒绝接收或转发软件的设计与编程方法。

1.5.12 基于特征码的恶意代码检测系统的设计与编程训练的基本内容与目的

基于特征代码的恶意代码检测系统是一种对文件进行扫描检测判定是否含有恶意代码的安全软件。这里在系统分析恶意代码检测系统基本工作原理的基础上,以基于特征检测的恶意代码检测系统为对象,研究恶意代码检测系统的设计与软件编程方法。

训练的主要目的是:

(1) 掌握恶意代码的分类、主要文件格式和相关检测技术等基本概念和背景知识。

(2) 掌握基于特征的恶意代码检测系统的基本工作原理、设计与实现方法。

(3) 掌握使用 p3scan 和 ClamAV 组建邮件病毒拦截网关的软件编程方法。

表 1-1 总结了以上 3 种类型的 12 个软件编程训练课题的选取思路、训练目的和要求。

表 1-1 编程训练的目的与要求

序号	类型	课 题	目 的
1	密码学及其应用	基于 DES 加密的 TCP 聊天程序	(1) 理解对称加密算法 DES 的基本工作原理 (2) 掌握将对称加密 DES 算法应用于网络通信的基本设计方法与实现技术 (3) 掌握 Linux 操作系统 socket 编程的基本方法
2		基于 RSA 算法自动分配密钥的加密聊天程序	(1) 理解 RSA 算法的基本工作原理 (2) 掌握将 RSA 算法应用于网络通信系统的基本设计方法与实现技术 (3) 掌握在 Linux 操作系统中实现 RSA 算法的编程方法 (4) 了解 Linux 操作系统异步 I/O 接口的基本工作原理
3		基于 MD5 算法的文件完整性校验程序	(1) 理解 MD5 算法的基本原理 (2) 掌握利用 MD5 算法生成数据摘要的计算方法 (3) 掌握将 MD5 算法应用于文件完整性校验的基本设计与编程方法 (4) 掌握在 Linux 操作系统中检测文件完整性的基本方法

续表

序号	类型	课　题	目　的
4	基本训练	基于 Raw Socket 的 Sniffer 系统	(1) 理解网络嗅探器 Sniffer 的基本工作原理与实现方法 (2) 掌握 Raw Socket 的基本工作原理 (3) 掌握 TCP/IP、ICMP 协议原理及 socket 编程方法
5		基于 OpenSSL 的安全 Web 服务器	(1) 理解 HTTPS 协议与 SSL 协议的基本工作原理 (2) 掌握使用 OpenSSL 编程的方法 (3) 掌握安全 Web 系统设计的基本设计与编程方法
6		网络端口扫描器系统	(1) 理解网络端口扫描器的基本结构、工作原理与设计方法 (2) 掌握 TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描，以及 UDP 扫描的基本工作原理、设计与实现方法 (3) 掌握 ping 程序的设计与编程方法 (4) 掌握 Linux 操作系统多线程编程的基本方法
7		网络诱骗系统	(1) 理解网络诱骗系统的基本工作原理 (2) 理解 Linux 系统调用实现和原理，以及对系统调用的扩展方法 (3) 掌握 Loadable Kernel Module 编程的相关知识和方法 (4) 了解 Linux 系统中程序隐藏的方法
8		入侵检测系统	(1) 掌握基于特征的入侵检测系统的基本工作原理、设计与实现方法 (2) 掌握 K-Means 算法的计算过程 (3) 掌握在网络安全研究中应用数据挖掘技术的基本概念与方法
9		基于 Netfilter 和 IPTables 防火墙系统	(1) 理解防火墙技术的基本工作原理 (2) 理解 Linux 环境中 Netfilter/IPTables 的工作机制 (3) 掌握对 Netfilter 内核模块进行扩展编程的基本方法 (4) 掌握通过 IPTables 构建防火墙的基本方法
10	综合练习	Linux 内核网络协议栈加固程序	(1) 理解 TCP 连接建立过程，以及拒绝服务式攻击的基本原理与方法 (2) 结合 Linux 内核源代码的分析，理解 Linux 网络协议栈的实现原理 (3) 掌握对 TCP SYN Flood 的防御手段，以及对 Linux 内核进行扩展开发的方法 (4) 了解 Linux TCP cookie 防火墙的工作原理
11		利用 Sendmail 收发和过滤邮件系统	(1) 掌握 Linux 环境下利用 Sendmail 邮件服务器的 milter 接口的回调函数实现简单的垃圾邮件过滤软件编程的基本思路和方法 (2) 掌握利用黑名单与白名单判断邮件的拒绝接收或转发软件的设计与编程方法 (3) 掌握根据关键字过滤方法判断邮件的拒绝接收或转发软件的设计与编程方法
12		基于特征码的恶意代码检测系统	(1) 掌握恶意代码的分类、主要文件格式和相关检测技术等基本概念和背景知识 (2) 掌握基于特征的恶意代码检测系统的基本工作原理、设计与实现方法 (3) 掌握使用 p3scan 和 ClamAV 组建邮件病毒拦截网关的软件编程方法

1.6 网络安全软件编程课题训练教学指导

1.6.1 网络安全软件编程训练课题选题的指导思想

创新是一个民族的灵魂，而在网络与信息安全领域培养具有创新能力的高水平人才，产生创新性研究成果，开发具有自主知识产权的产品尤为重要。当前我国网络安全课程的教学水平还远不能够满足国家信息化建设高速发展的要求。在网络安全教学过程中，教学内容与当前技术的发展水平、理论教学与实际工作能力的培养差距明显。这些问题将严重地制约网络安全人才的培养质量与产业的发展。

从事网络安全与信息安全专业的技术人员可以分为工程师、高级工程师与专家等多个层次，社会对各个层次人才的需求都非常强烈。大学教育如果只能培养使用网络安全产品的人是远远不够的。现在我国网络安全产品的研发很多是在国外公开的网络安全开源软件的基础上进行的。这种方法从表面上看是一条捷径，"立竿见影"，可以很快见成效，但是我们不能不清醒地认识到这是一种"短视"行为，并且存在巨大和潜在的危机，对于要真正形成具有我国自主知识产权的网络安全产品非常不利。大学在对高层次信息安全专门人才的培养上必须要正视这个问题，要下苦功夫从基础开始，培养能够产生创新思想与具备研发能力的专门人才。

1.6.2 网络安全软件编程训练课题选题覆盖的范围

为了达到上述目的，作者结合多年的科研与教学实践，总结出 12 个"近似实战"的研发课题。

网络安全训练的课题覆盖了从密码学在网络通信与数据安全中的应用，网络端口扫描、网络嗅探器、网络诱骗、网络入侵检测、安全 Web、防火墙、Linux 内核网络协议栈程序加固，到网络病毒与垃圾邮件的检测与防治技术，训练课题接近学科研究的前沿，覆盖了网络安全研究的主要方向。软件编程训练课题可以分为：密码学及应用、网络安全常规技术、当前研究的热点课题的综合训练等 3 个部分，训练课题的比例是 3∶6∶3。

课题内容覆盖了以下 3 个方面的问题：

(1) 网络安全软件设计中涉及的基本问题与基本方法。

(2) 网络安全软件编程中需要使用的基本工具。

(3) 网络安全技术发展中的热点问题与解决思路。

1.6.3 网络安全软件编程训练课题编程环境的选择

完成网络安全训练课题的操作系统选用 Linux，目的是希望充分利用 Linux 开源软件的优势，通过在 Linux 环境中完成网络安全软件的设计与编程训练，增强读者研发具有自主知识产权的技术与产品的能力。完成本书的训练课题不需要限定任何特殊的硬件环境和编程语言。

Linux 作为应用最广泛的开源操作系统之一，不仅能够提供终端主机所需要的各种网络协议软件，而且还能够实现网桥、路由器等网络设备的基本功能。

为了帮助读者尽快掌握 Linux 环境中网络安全软件设计与编程方法，本书的第 2 章从两个方面对 Linux 网络协议栈进行讨论：

(1) 结合 Linux 网络协议栈的设计特点，介绍了网络协议栈源码所包含的几个主要功能模块，其中包括：路由子系统、组播模块、IPv6 模块、包过滤和防火墙模块、邻居子系统、网桥模块、流量控制管理、原始套接字和 PACKET 协议族等。

(2) 讨论了 Linux 网络协议栈源码发送和接收 TCP 报文的基本流程。

有兴趣的读者可以在此基础上，通过进一步阅读 Linux 网络协议栈的源码，学习其他的功能模块。

1.6.4　网络安全软件编程训练选题指导

训练选题的建议考虑到不同基础的读者选择课题时涵盖的知识点内容，包括网络训练涉及的层次、课题类型、编程的难度等因素。作为教材使用时，任课教师可以结合教学内容的需要和学时灵活地指导选择训练课题。自学的读者可以根据自己的基础和兴趣，结合训练选题的建议考虑在不同的阶段，选择不同的课题，循序渐进地组织学习，逐步提高网络软件的编程能力。

从研究生和高级人才培养的角度，应该强调“研究型”与“自主型”的学习方式。对于研究生教学过程来说，学生应该变被动的“听课、做笔记”转向主动、研究地学习和提高。从任课教师与导师的角度应该强调“因材施教”。不同基础和不同需求的读者可以根据个人的基础、学习与工作的需要，选读其中的某些章节，完成其中部分课题的编程任务。

研究生教材不应该只是一本一学期使用的教科书，更应该是一本技术参考书，甚至是一本手册。导师可以根据需要选择教材中的部分内容，作为基本的学习要求。学生学习的过程应该在导师的指导下有选择地自学和阅读，完成编程训练。有些内容可能第一次仅仅是读过和了解，如果在今后的科研、开发工作需要，可以再回过头来继续阅读和参考。

作为研究生与网络安全高级人才能力培养的教材，希望读者能够在阅读书中相关章节之后，独立地完成课题的编程任务。从严格训练的角度出发，书中只提供了解决问题的思路，给出了启发读者的编程示例，书后所附的光盘中给出了编程所需要的编程工具与测试数据集，希望读者通过阅读相关的章节，结合自己已经掌握的网络知识与基本编程方法，独立地完成训练课题的要求，通过下苦功夫、扎扎实实地训练，深入理解理论知识，提高实践能力，使研究生与从事网络安全工作的工程技术人员在学习过程中体会到“研究型”与“自主型”学习的快乐。

第 2 章

Linux网络协议栈简介

Linux 作为应用最广泛的开源操作系统之一，不仅能够提供终端主机所需要的各种网络协议软件，而且还能够实现网桥、路由器等网络设备的基本功能。为了达到培养读者研发具有自主知识产权网络安全软件产品的目的，本书要求所有的编程训练在 Linux 操作系统的基础上进行。本章将从两个方面对 Linux 网络协议栈进行讨论：一是结合 Linux 网络协议栈的设计特点，介绍网络协议栈源码包含的几个主要功能模块；二是讨论 Linux 网络协议栈源码发送和接收 TCP 报文的基本流程。有兴趣的读者可以在此基础上，进一步阅读 Linux 网络协议栈的源码来学习其他的功能模块。

2.1 Linux 网络协议栈概述

Linux 网络协议栈源码是 Linux 内核源码的重要组成部分，它不仅支持 TCP/IP 协议，还支持 Ethernet 透明网桥协议，提供 IP 防火墙、IP 服务质量(QoS)管理及其他的安全特性。配置好的 Linux 系统所能提供的这些功能可以与目前使用的中档路由器、网桥等网络设备相媲美。Linux 系统还支持多种不同的网络协议，如 ATM、蓝牙协议等。本章将重点讨论基于 Ethernet 的 TCP/IP 协议的设计和实现技术。

2.1.1 Linux 网络协议栈的设计特点

Linux 网络协议栈在设计和实现思路上体现了以下 3 个重要特点：

1. 层次化

网络系统在设计上采用了层次化的体系结构。这种层次结构为网络协议的设计与实现提供了很大方便，上层协议可以通过相邻层之间预先定义的服务接口直接使用下层协议提供的功能，从而保证任何一层协议实现技术的改变不会影响整个网络系统的正常运行。由于操作系统不仅要考虑网络子系统如何与其他子系统(如文件子系统和调度子系统)协调工作，还要支持各种不同的网络设备，让系统中的各个部分协调工作，所以必须采用层次化的设计方法。Linux 网络协议栈的层次结构如图 2-1 所示。

(1) 网络编程接口层

位于层次结构最高层的是网络编程接口层，它主要提供符合 BSD socket API 规范的接口函数，这部分就是编写网络应用程序时经常采用的 socket 函数，例如 socket、bind、accept 等，它们是应用程序使用网络服务的主要途径。

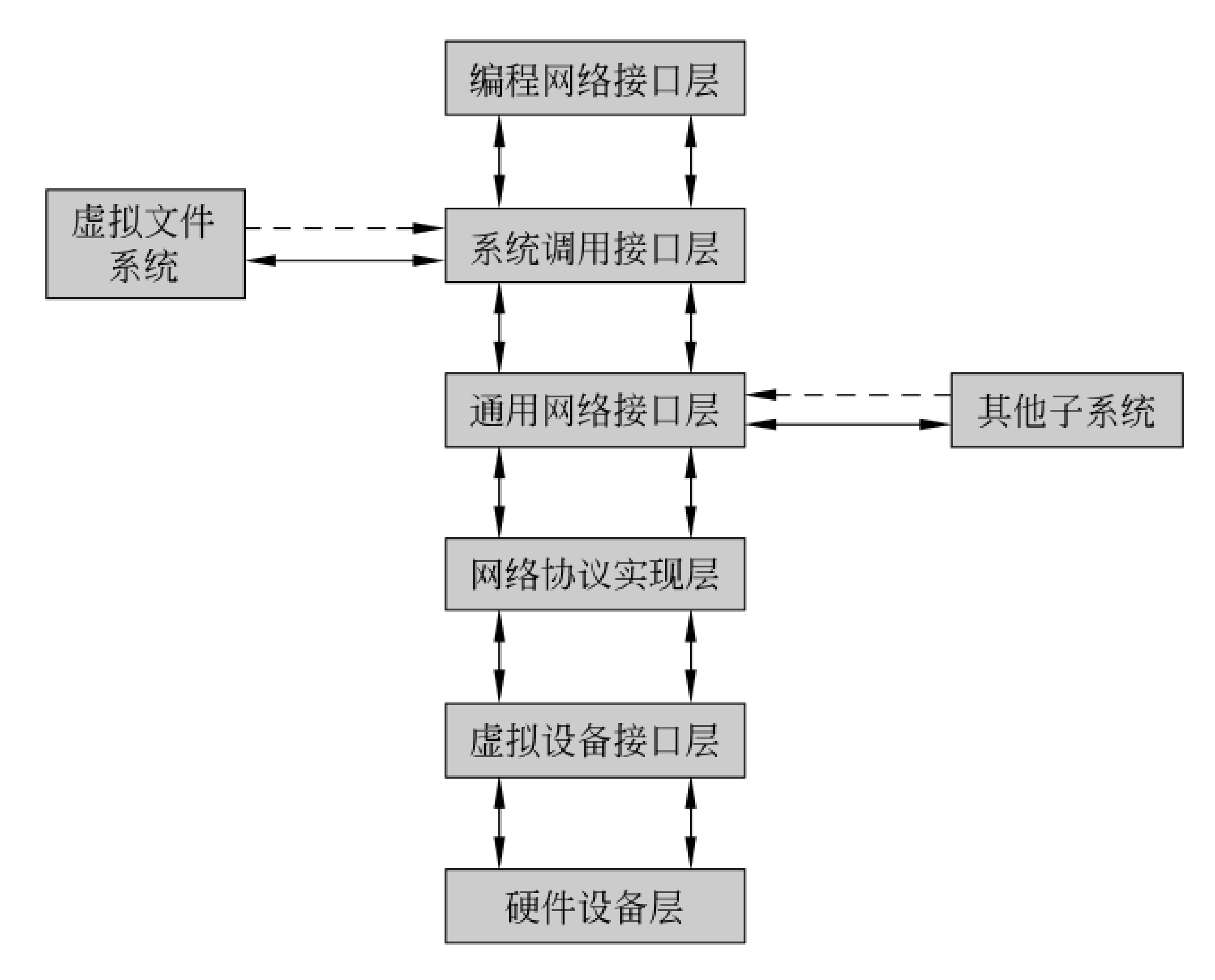


图 2-1　Linux 网络协议栈的层次结构示意图

(2) 系统调用接口层

在网络编程接口层之下的是系统调用接口层，它可以实现通过虚拟文件系统的接口访问网络的功能，如在报文发送和接收流程中使用的 read 和 write 函数。

(3) 硬件设备层

位于层次结构最底层的是由各种具体硬件设备组成的硬件设备层，各种设备通过硬件厂商自己提供的驱动程序来完成工作。

(4) 虚拟网络设备层

位于硬件设备层之上的是虚拟网络设备层，Linux 中采用 net_device 结构来抽象地表示系统中的每一个网络硬件设备。struct net_device 中定义了所有网络设备都必须保存的信息和必须支持的操作。虚拟网络设备层还可以看做是一个适配层，其作用类似于面向对象设计中的接口，它可以屏蔽底层各种硬件本身的差异，使得上层可以通过一个统一的接口来操作各种网络设备，完成报文接收与发送任务。

(5) 网络协议实现层

虚拟网络设备层之上是网络协议实现层，其中实现各种网络协议，如 TCP、IP、ARP 协议等。

(6) 通用网络接口层

Linux 网络协议栈在网络协议实现层之上增加了一个通用网络接口层。设置通用网络接口层的目的与虚拟网络设备层十分类似，它为各种网络协议提供了一个服务接口。这样系统中的其他子系统就可以直接使用网络服务，而无需依赖特定的协议或硬件。

2. 模块化

一方面，层次化设计本身就要求把每一层作为一个独立的模块来实现；另一方面，由于 Linux 网络协议栈支持的网络协议和实现的网络功能比较多，所以通常情况下这些功能或协议不会同时被使用，这就要求网络协议栈能够根据实际情况，采取“按需加载”的方法，选

择所要求的功能与协议。通过采用模块化的实现方式,不仅能够提高系统的工作效率,而且便于以后增加新的功能或协议。图 2-2 给出了 Linux 中 TCP/IP 协议栈中的各个主要功能模块之间的关系示意图。

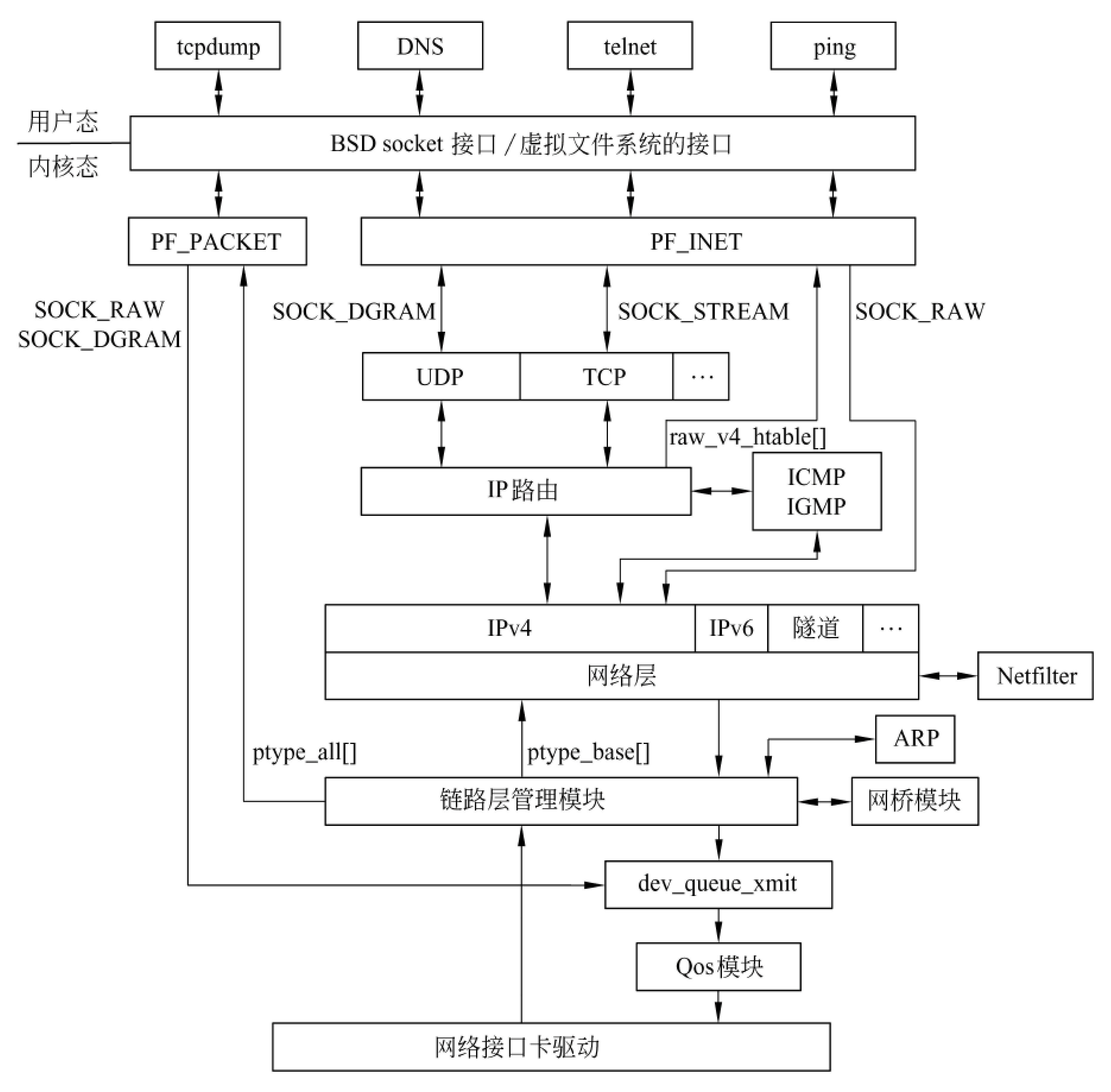


图 2-2 Linux 中 TCP/IP 协议栈的主要功能模块关系示意图

每个模块的设计目标都很明确,一个模块只用来完成一项任务,如 IPv4 模块只完成 IPv4 分组的接收、发送、合法性判断、分片重组等任务,而路由选择功能则交给 IP 路由模块来完成。为降低模块之间的耦合度,减小相互之间的影响,模块与模块之间的界面多采用函数指针作为接口。这样,当一个模块的实现方式发生变化时,不会影响另一个模块的正常工作。

3. 面向对象的设计

Linux 内核里很多子系统都采用了面向对象的设计方法,最典型的例子就是虚拟文件系统(virtual file system,VFS)。Linux 内核支持很多种文件系统,比如 VFAT、EXT2、JFS 等。但是在内核里面,采用一个抽象的基类 VFS 来表示所有的文件系统,并且定义了文件系统的操作界面(接口函数),然后每一种文件系统再根据 VFS 进行实例化。

网络协议栈部分也有多处采用了面向对象的设计方法,典型的例子有以下两个:

（1）邻居表

ARP 协议是获取网络结点 IP 地址与 MAC 地址映射关系的协议。但在其他协议族（如 ATM、X25）中还存在另外几种不同的地址映射关系。Linux 网络协议栈对此进行了适当的抽象，采用“邻居”的概念来管理相邻的计算机，这样不同协议族都采用相同的接口来管理邻居。

（2）socket 接口

由于 Linux 网络协议栈支持 20 多种协议，因此它在内核中采取向用户层提供一个统一的 BSD socket 接口的方法。这样做的最大优点是允许应用程序在利用不同的网络协议通信时，能够采用相同的函数完成数据的发送与接收。

采用面向对象设计方法的主要目的是能够获得多态特性。这种设计在实现上采用函数指针，同样一个调用形式，被执行时会根据指针赋值情况的不同而调用不同的函数，从而表现出不同的行为。当需要支持新的协议时，只需要提供实现了所需功能的处理函数，然后让函数指针指向新协议的处理函数即可，这样便可以在不需要修改调用函数代码的情况下，动态增加对新协议的支持。这种设计方式极大地提高了代码的可扩展性和灵活性。

需要注意的是：灵活性与复杂性总是相伴而生的。采用函数指针作为函数调用的形式之后，会给阅读源代码、追踪和分析程序的处理流程带来很大困难，即无法从调用语句本身确定被调用的函数。因此，在本章分析报文发送和接收流程时，会特别强调代码执行过程中对函数指针的处理情况。

2.1.2 Linux 网络协议栈代码中使用的固定实现模式

除了上面介绍的 3 个设计特点之外，Linux 网络协议栈的代码中还使用了很多固定的实现模式，下面将要介绍几个与本章内容相关的模式。

1. 缓存

网络协议栈为了提高处理效率而使用缓存，具体的实例包括保存路由结果的 struct rtable 结构和 ARP 缓存等。通常使用哈希表来实现缓存，内核中提供了用于实现哈希表的基本数据结构：数组、单向和双向链表。具体的哈希函数要根据缓存对象的特征来进行选择，有些情况下会在键值中加入一些随机特征来防止针对哈希表的拒绝服务（Denial of Service，DoS）攻击。

2. 引用计数

内核中的很多数据结构都可能被不同 CPU 上的不同进程所共享，这里就涉及如何进行垃圾收集的问题，即只有不被任何进程使用的数据结构才能被释放，否则就会引起空指针等严重的问题。因此网络协议栈中的很多数据结构都使用了一个引用计数字段。使用该结构的用户在使用之前增加引用计数的值，在使用之后再减少计数值。当引用计数值减少到 0 时，就表明这块数据已经不再有用户使用，应该释放其占用的内存。在 Linux 内核源代码中包含引用计数字段的数据结构往往会同时提供两个专门的函数来增加和减少计数值，这类函数的名称通常为：xxx_hold 和 xxx_release（或 xxx_put，例如 net_device 结构提供的

dev_put 函数)。如果在释放数据结构时忘记调用 xxx_release 函数减少计数值,则可能造成数据永远得不到释放,导致内存泄露;如果在使用数据结构时忘记调用 xxx_hold 函数来增加计数值,则可能导致当前使用的数据被提前释放,造成空指针问题。特别需要指出的是,在使用哈希表或链表提供的查询函数获取其中元素时,查询函数往往会自动增加该元素的计数值,所以在使用完之后不能忘记手工减少计数值。

3. 函数指针

通过在结构体中定义函数指针成员,C 语言也可以编写出具有面向对象特性的程序。使用函数指针的最大优点是可以根据情况将指针初始化为不同的函数,从而做到同一种静态调用形式具有不同动态行为的能力,即多态性。在 Linux 网络协议栈中使用函数指针的情况主要有 3 种。第一,作为层与层之间的接口实现一对多的映射关系,如 BSD socket 层和具体协议族的 socket 实现之间就借助于一组函数指针(定义在 proto_ops 结构中),获得接口和多态的特性;第二,根据状态或其他模块的处理结果来选择具体的处理函数,如 ARP 模块的发送函数指针 output 会根据 ARP 缓存的状态来决定采用哪一种发送函数;第三,实现一些自定义的处理,例如在 net_device 中定义的函数指针 init 就可以执行设备提供的自定义初始化函数来完成特殊处理,无需特殊处理时函数指针为空,这种使用方法通常以下面代码的形式出现:

```
if (dev->init && dev->init(dev) != 0) {
        ...
    }
```

函数指针的最大缺点是会给阅读源代码带来困难。当在某条代码执行路径上遇到函数指针时,必须首先找出对该指针的初始化情况。常见的初始化条件包括某些报文首部字段或状态,例如,在将报文投递给上层处理函数时,会根据报头中上层协议字段的值来初始化函数指针;在 ARP 模块中则会根据缓存状态让函数指针指向合适的处理函数。

2.1.3 TCP/IP 协议栈中主要模块简介

从图 2-2 中可以看出,网络协议栈的所有功能模块都在内核态中完成,这些内核模块相互协作实现了多种多样的网络功能。

这些网络功能通过 BSD socket 接口和虚拟文件系统的接口(主要针对 socket 的读写操作)提供给用户态的程序使用,这一部分就是大家熟悉的 socket 编程函数。在使用 socket 进行网络编程时,必须要指明所使用的网络协议族才能创建 socket,图 2-2 中给出了两种具体的协议族:PF_INET 和 PF_PACKET。PF_INET 表示 Internet 上使用的是 TCP/IP 协议族,该协议族提供 3 种服务类型,即 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAW。前两者分别对应着 TCP 和 UDP 协议,而 SOCK_RAW 类型的 socket 则会直接将用户提供的数据交给 IP 协议进行封包,即位于 IP 层之上的各层报头都必须由用户自行构造。TCP 和 UDP 协议是 PF_INET 协议族中两种主要的传输层协议,在内核中分别由两个独立的模块来实现报头的构造以及发送和接收流程的处理。

下面按照从上层到下层的顺序来对网络协议栈的各个子模块进行介绍。

1. 路由子系统

路由是 IP 层的主要任务之一，也是 IP 层最复杂的模块。当主机向外发送 IP 分组时，要根据 IP 分组的目的地址查询它的路由。Linux 网络协议栈将与 IP 路由有关的信息集中存放在转发信息块(Forwarding Information Block,FIB)中，转发路由信息库包括路由规则(Routing Rules)和路由表(Routing Tables)两个部分。路由规则用来实现基于规则的路由，又称为策略路由，即对满足某些条件的报文，应用特定的路由表来决定路由信息。系统中最多可以定义 255 张路由表，在不启用策略路由的情况下只使用其中两张：主路由表(main_table)用于描述主机间路由规则；局部路由表(local_table)只用于记录本地地址信息。如果分组的路由决策来自局部路由表，则说明这是发给本机的分组。通过这样一个特殊的路由表，路由子系统能够以统一的方式来处理所有 IP 分组。

路由表本身不是一个单一的结构，而是由多个结构组合而成，并且采用层次结构来组织和管理这些实现路由表的结构。这些数据结构之间的关系如图 2-3 所示。Linux 网络协议栈用 struct fib_table 结构来表示一张路由表。对每一张路由表来说，首先根据子网掩码(netmask)的长度(0～32)将所有目的网络分成 33 个组，每个组用 struct fn_zone 来表示，然后在同一子网掩码(同一组)中，再根据子网的不同(如 192.168.1.0/24 和 192.168.2.0/24)，划

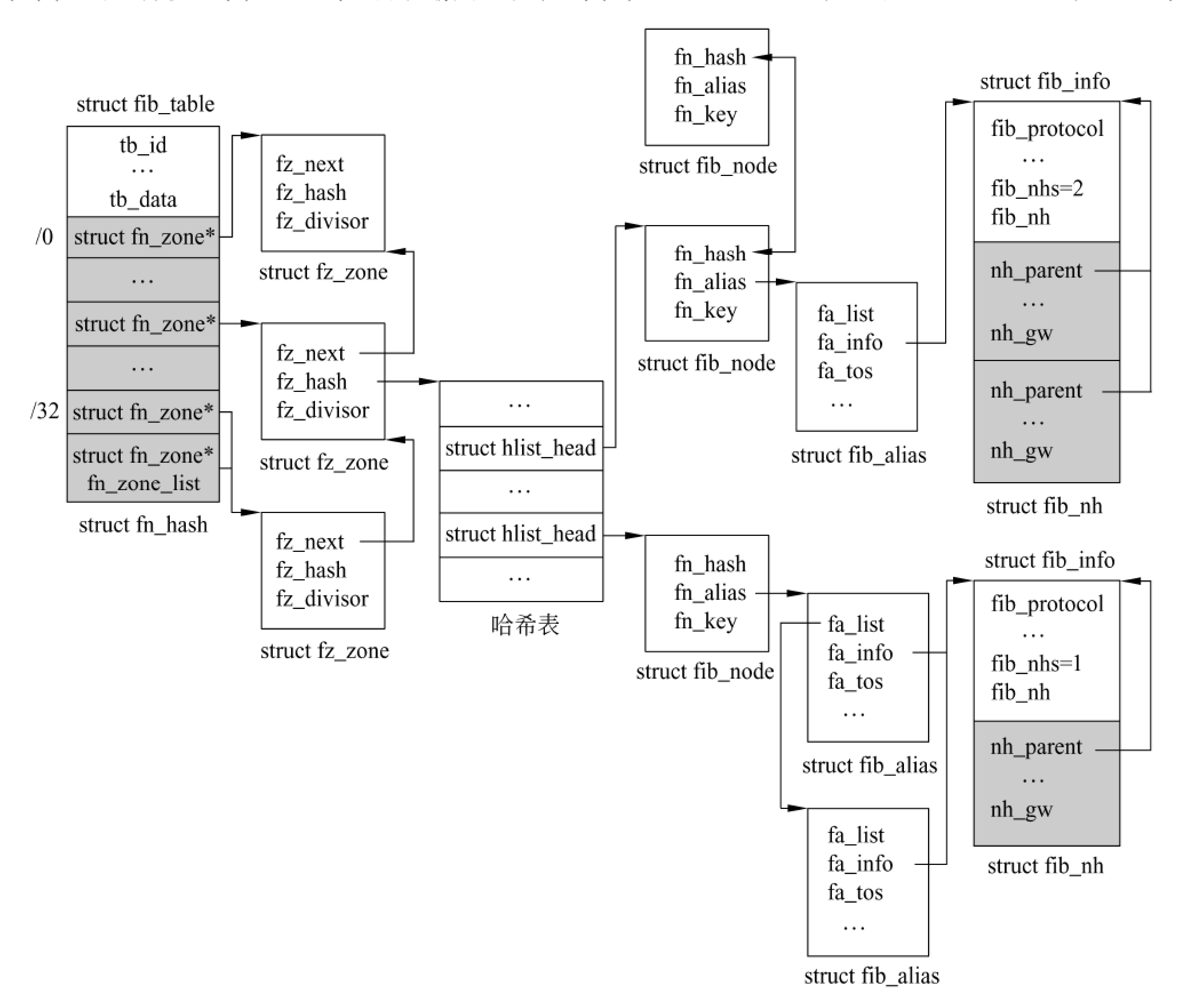


图 2-3 实现 IP 路由表的数据结构示意图

分为第 2 层，用 struct fib_node 结构来表示，对于同一个目的子网，可能由于 TOS 等属性的不同而使用不同的路由，这就是路由表中的第 3 层，用 struct fib_alias 来表示。第 3 层结构表示一个路由表项（routing entry），而每个路由表项都要包含其他一些参数，例如协议类型、下一跳地址等，这些信息保存在 struct fib_info 结构中。其中最重要的下一跳信息由 struct fib_nh 结构来表示（nh 表示 next hop），它总是附属在对应的 fib_info 结构之后。同一个目的网络可能有多个下一跳地址，即多路径路由（multipath routing），因此一个 fib_info 结构中可能对应多个 fib_nh 结构。分层结构的优点显而易见，它使路由表的实现更加灵活高效，逻辑上也更加清晰，并且可以方便地共享路由信息（如 struct fib_info），从而减少了数据的冗余。

路由子系统对外提供服务的接口是 fib_lookup 函数，它会根据是否启用策略路由等条件，找到合适的路由表，然后通过 fib_table 结构中的函数指针 tb_lookup 调用 fn_hash_lookup 函数来查找需要的路由表项。

由于 FIB 表的结构比较复杂，通过它查找路由是一个比较“缓慢”的过程，所以 Linux 网络协议栈中还实现了路由缓存模块，将经常使用的 FIB 信息缓存起来，以达到加快路由查找速度的目的。为了实现路由缓存，Linux 在 include/net/dst.h 中定义了 struct dst_entry 结构，它表示一个与具体协议无关的路由缓存。对于 IP 协议来说则在 include/net/route.h 中定义了 struct rtable 结构来表示 IP 路由缓存。rtable 结构的开头部分定义如下：

```
union {
    struct dst_entry dst;
    struct rtable * rt_next;
} u;
```

一方面，联合体 u 要么表示 dst_entry 结构，要么指向另一个 rtable 结构。另一方面，dst_entry 结构的第一项是指向另一个 dst_entry 结构的 next 指针，而且 Linux 网络协议栈不会直接创建 dst_entry 结构，而是通过创建 rtable 结构，间接生成 dst_entry 结构，所以指向 rtable 和 dst_entry 结构的指针可以自由进行类型转换。

分组的发送过程中需要通过路由子系统来确定下一跳的地址，在内核中完成这一功能的是 ip_route_output_flow 函数，它会选择或创建对应的 rtable 结构来作为路由的结果。在分组接收过程中同样需要使用路由子系统确定是接收还是转发分组，这一步由 ip_route_input 函数来完成，网络协议栈会根据路由的结果决定下一步调用的处理函数。

2. 组播模块

Internet 上的大多数网络应用都属于单播通信，即将分组从一台主机发送到另一台主机。但有些应用（如视频会议）要求同时有多个发送主机和多个接收主机，这样的通信方式称为组播（或多播）。上一小节介绍的 IP 路由子系统主要是针对单播模式，但 Linux 网络协议栈同样也支持组播模式。要实现组播通信，网络协议栈必须实现两种功能：管理组播通信的成员、高效地把数据分发给组播成员。

对于第一种功能，Linux 协议栈实现了标准的 IGMP 协议，具体的代码在 net/ipv4/igmp.c 中。IGMP 协议数据必须作为 IP 分组的负载，在 IP 报头中会指明负载类型，进而调用接收 IGMP 报文的处理函数 igmp_rcv。

对于第二种功能，通常在组成员之间建立一棵组播树，发送主机作为树的根节点，把分组发送给作为树的叶子节点的接收主机，这个过程称为组播路由算法。树的中间节点只负责转发而不需要接收组播分组，因此网络协议栈也必须区分接收和转发的情况。目前常用的组播路由算法包括 DVMRP，MOSPF 等。组播路由算法大多以守护程序的形式在用户空间实现。但转发和接收组播报文的处理必须在内核协议栈中完成，这部分代码在 net/ipv4/ipmr. c 中，主要的处理函数是 ip_mr_input、ip_mr_forward 和 ipmr_queue_xmit。

3. IPv6 模块

IPv6 将地址长度从 32bit 扩展为 128bit，同时其在报头格式、报头扩展选项等方面与 IPv4 存在差异。Linux 网络协议栈中实现 IPv6 的代码保存在 net/ipv6 目录下及头文件 include/net/ipv6. h 中。IPv6 协议的代码以 IPv4 的实现为基础，尽可能复用一些与具体 IP 协议版本无关的处理代码。

分组要进入 IPv6 模块只有以下 3 种途径：

(1) 在处理数据链路层报头时发现网络层使用 IPv6 协议，调用函数 ipv6_rcv 来接收并继续处理该报文；

(2) 当上层协议(如 TCP 或 UDP)发送报文时，调用 IPv6 的发送函数 ip6_xmit；

(3) 通过 IPv6 来发送 ICMP 报文时调用 icmpv6_send 函数。

IPv6 同样也需要完成转发报文的工作，这是由 ip6_forward 函数完成的。另外在 net/ipv6/ip6_fib. c 中实现了针对 IPv6 地址的路由表管理，在 net/ipv6/tcp_ipv6. c 中实现了在 IPv6 条件下 TCP 协议必须进行的修改。

4. 报文过滤和防火墙模块

Linux 使用 Netfilter 框架来实现防火墙的功能。从 Linux2. 4 内核开始，内核设计者在网络协议栈中预留出若干函数接口，开发人员可以通过这些接口实现报文过滤、报文处理、NAT 等功能，这套函数接口构成了 Netfilter 框架。Netfilter 框架包含以下 3 个部分：

(1) 为每种网络协议(IPv4、IPv6 等)定义一套钩子函数(IPv4 定义了 5 个钩子函数)，这些钩子函数在报文流过协议栈的几个关键点时被调用。在这几个钩子函数点上，协议栈将把报文及钩子函数标号作为参数传递给 Netfilter 框架。

(2) 内核中的任何模块都可以在每种协议的一个或多个钩子点上进行注册，实现挂接。这样当某个报文经过 Netfilter 框架设定的检查点时，内核就会检测是否存在某些模块对该协议和钩子函数进行了注册。若存在，则调用该模块在注册时提供的回调函数。通过这些回调函数，其他模块就有机会检查(可能还会修改)报文、丢弃该报文或者将该报文传入用户空间的队列等待进一步处理。

(3) 用户进程可以采用异步方式处理传递到用户空间的报文。当用户态所需操作完成之后，还可以重新将该报文注入到内核中对应的钩子函数点上，继续进行网络协议栈的处理流程。

Linux 中所有报文过滤和 NAT 等功能都基于该框架。目前 Netfilter 框架已在 IPv4 和 IPv6 协议栈中实现了。

图 2-4 显示了 Netfilter 框架中钩子函数在协议栈中的位置，从图中可以看到 IPv4 共有

5 个钩子函数点。主机接收到的报文在进行 IP 校验后，会经过第一个钩子函数点 NF_IP_PRE_ ROUTING 进行处理；然后进入路由代码，决定该报文是需要转发还是由本机接收；若该报文是发给本机的，在经过钩子函数点 NF_IP_LOCAL_IN 上的处理之后，会传递给上层协议；若需要转发该报文，则会进行 NF_IP_FORWARD 点上定义的处理；转发的报文通过最后一个钩子函数点 NF_IP_POST_ROUTING 上的处理以后，才能传输到网络上。本地产生的报文经过钩子函数 NF_IP_LOCAL_OUT 处理后，进行路由选择处理，然后经过 NF_IP_POST_ROUTING 处理再发送到网络上。

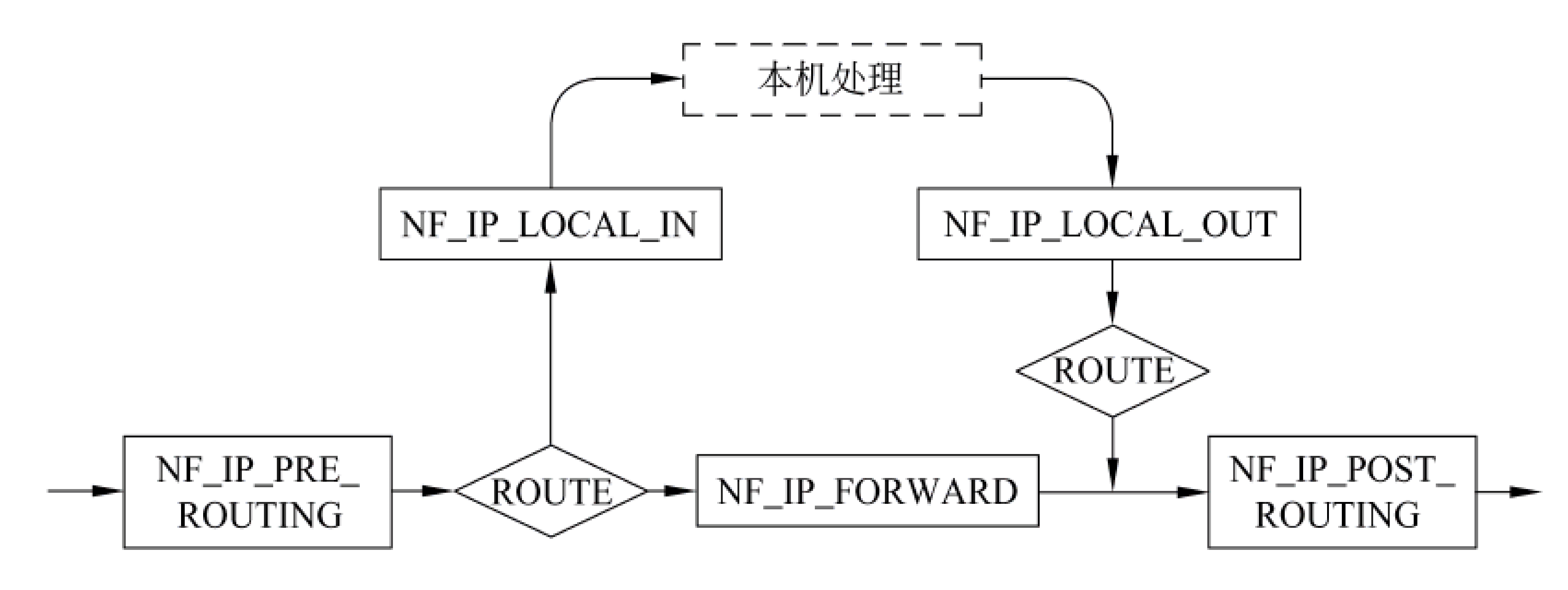


图 2-4 Netfilter 框架中钩子函数的位置示意图

Iptables 是一个基于 Netfilter 框架的报文过滤和修改工具。Iptables 模块可以创建规则表(table)，并要求报文流经指定的规则表。在 Iptables 中，预先定义了 3 张规则表，分别实现 3 种不同的概念：报文过滤(filter 表)、网络地址转换(nat 表)及报文处理(mangle 表)。

(1) 报文过滤(packet filtering)

filter 表不会对报文进行修改，而只对报文进行过滤。它是通过钩子函数点 NF_IP_LOCAL_IN、NF_IP_FORWARD 及 NF_IP_LOCAL_OUT 接入 Netfilter 框架的。因此对于任何一个报文都有且仅有一个地方对其进行过滤。

(2) 网络地址转换(NAT)

nat 表在 3 个 Netfilter 钩子函数点上进行回调函数的注册：NF_IP_PRE_ROUTING、NF_IP_POST_ROUTING 及 NF_IP_LOCAL_OUT。NF_IP_PRE_ ROUTING 实现对需要转发的报文的源地址进行地址转换，而 NF_IP_POST_ROUTING 则对需要转发的报文的目的地址进行地址转换。对于本地报文的目的地址的转换则在 NF_IP_LOCAL_OUT 点上实现。nat 表格不同于 filter 表格，因为只有新连接的第一个报文会遍历表格，而随后的报文将根据第一个报文的结果进行同样的转换处理。nat 表可以实现源 NAT、目的 NAT、伪装(源 NAT 的一个特例)及透明代理(目的 NAT 的一个特例)。

(3) 报文处理(packet mangling)

mangle 表在 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_OUT 钩子中进行注册。使用 mangle 表，可以实现对报文的修改或给报文附加一些带外数据。当前 mangle 表支持修改 TOS(Type Of Service)位，以及设置 sk_buff(Linux 内核中表示报文的数据结构)的 nfmark 字段等。

每一张规则表都在指定的 Netfilter 钩子上注册了若干个回调函数，当报文到达这些钩子时，就会进入相应的函数进行处理。实际上，在这些函数中，将会逐一匹配并执行若干条由管理员制定的防火墙规则(rule)，这些规则组成了一个规则链。因此，整个 Iptables/

Netfilter 系统可以形象的看成：Netfilter 是表的容器，表是链的容器，链是规则的容器。

每一条规则的定义形式为："如果报头符合某个条件，就按照某种方法处理这个报文"。例如"如果报文的目的地址是 192.168.1.254，就丢弃该报文"。当一个报文到达一个规则链时，系统就会从第一条规则开始检查报文是否符合该规则所定义的匹配条件(match)。如果满足，系统将根据该条规则所定义的目标方法(target)处理该报文；如果不满足则继续检查下一条规则。最后，如果该报文不符合该链中任何一条规则的话，系统就会根据该链预先定义的策略(policy)来处理该报文。防火墙的用户/管理员可以使用 Iptables 命令及其选项来管理防火墙，设置防火墙规则。

在 Linux 内核中实现 Iptables/Netfilter 框架的源代码保存在 net\netfilter\ 和 net\ipv4\netfilter\ 目录下。

Linux 网络协议栈中实现的防火墙不仅可以实现报文过滤，还能够根据连接情况将进出网络的数据识别为不同的会话，针对每个会话建立状态连接表，利用状态表跟踪每一个会话的状态。基于状态检查的防火墙不仅根据规则表对报文进行处理，还会考虑报文是否符合会话所处的状态，相对于报文过滤机制具有更强的识别和处理能力，更高的安全性。Linux 网络协议栈的这一特性通过连接跟踪(connection tracking)机制来实现。

连接跟踪就是跟踪每一条连接。连接跟踪机制使用在 linux/netfilter_ipv4/ ip_conntrack.h 中定义的 struct ip_conntrack 结构来描述一条连接。其他的实现代码保存在 net/ipv4/netfilter/ip_conntrack_standalone.c 和 net/ipv4/netfilter/ip_conntrack_core.c 文件中。这一功能模块的实现也基于 Netfilter 提供的钩子函数。如图 2-4 所示，连接跟踪模块在 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_OUT 上定义了高优先级的处理函数，这两个点分别是报文接收和发送流程中经过的第一个钩子点，因此报文在进入防火墙之前首先会进行连接识别处理，这一步会为每个报文找到所属的连接。此后，防火墙不仅能够看到单个报文的内容，还掌握了该报文所属连接的相关状态信息。防火墙可以根据这些连接的状态制定更加丰富的规则，以实现状态检测。

5. 邻居子系统

在通过网络转发分组的过程中，一台主机需要将分组转发给距离目标主机更近的相邻主机，并且需要知道相邻主机第 3 层和第 2 层地址之间的映射关系。在 TCP/IP 协议族中完成上述功能的是在 IPv4 中使用的 ARP 协议和在 IPv6 中使用的邻居发现(neighbor discovery)协议。

虽然不同的网络协议会采用不同的方法或协议来解决地址映射问题，但它们的目的是相同的，因此 Linux 的邻居子系统把这些公用部分抽象出来，形成一个与具体协议无关的服务接口，针对各种具体协议的实现则作为底层细节被隐藏在统一的接口之下。公用部分提供的主要功能包括：缓存机制和超时机制。其中缓存机制包括两个方面：第 3 层和第 2 层地址的映射关系缓存和第 2 层帧头信息(frame header)的缓存。各种映射关系都存在实效性问题，因此公用部分提供了一种通用的老化和超时机制来维护映射关系的有效性，并且定义了通用的状态迁移图。

在 Linux 网络协议栈的实现中，include/net/neighbour.h 文件中定义了一个相当于面向对象术语中抽象基类的结构 struct neigh_table，而某种具体的地址解析模块(如 ARP)则

是 neigh_table 结构的一个实例。例如一个 Linux 主机可能同时运行 IPv4 和 IPv6 两种网络层协议，它对应的 neigh_table 结构就是 arp_tbl 和 nd_tbl。表示一个具体邻居的结构是同样定义在 include/net/neighbour.h 中的 struct neighbour，它保存了每个邻居的具体状态数据、操作函数集和关键字等信息。在 neigh_table 中使用哈希表来维护由 neighbour 结构表示的邻居信息。相应的基本操作包括增加、查找、删除邻居表和邻居、维护邻居状态等，这些代码都定义在 net/core/neighbour.c 文件中。另外在文件 include/net/netdevice.h 中定义的 struct hh_cache 表示第 2 层帧头信息的缓存。

ARP 协议的实现代码保存在文件 include/net/arp.h 和 net/core/ipv4/arp.c 中。在 neigh_table 结构中定义了一个函数指针 constructor，它是公有部分留给具体协议按照自己的需要来初始化 neighbour 结构的一个机会。对于 ARP 协议来说，会利用这个机会调用 arp_constructor 函数，该函数主要完成对 neighbour→ops 的设定。一般情况下，这一组函数指针会指向全局变量 arp_hh_ops。当协议栈需要获得一对新的 IP 和 MAC 地址映射关系时，会通过函数指针间接调用 arp_solicit 函数来构造所需的 ARP 报文。负责发送和接收 ARP 报文的接口分别是 arp_send 和 arp_rcv 函数。

6. 网桥模块

网桥是一种工作在第 2 层的网络互联设备，主要用于在数据链路层上将两个或多个物理上独立的局域网连接为一个网段。网桥和交换机在本质上是相同的，前者主要用于在文档中(特别是 IEEE 规范)描述第 2 层连接设备和说明支撑树协议(STP)算法，而后者多指实际的网络设备。

网桥的主要功能是根据链路层的地址，如 Ethernet 中的 MAC 地址来进行帧转发。同时，网桥还具有自动学习功能，可以在帧转发的过程中智能地形成转发表。网桥的另一个重要特征是必须支持支撑树协议，以去除网络中可能出现的环路。网桥中最常用的是透明网桥。Linux 网络协议栈中实现了透明网桥的功能。

网桥设备在 Linux 网络协议栈中属于虚拟设备，它必须借助于物理设备来完成实际数据的收发。因此在启用网桥之前必须要将某些物理设备，如网络接口卡作为端口绑定到网桥上。

Linux 网桥部分的实现代码保存在 net/bridge/目录下。代码中定义 net_bridge 结构表示网桥设备，定义 net_bridge_port 结构来表示网桥上的一个端口，它们的结构声明都在 net/bridge/br_private.h 文件中。在文件 net/bridge/br_if.c 中提供了创建和删除网桥的方法：br_add_bridge 和 br_del_bridge 以及为网桥添加和删除端口的方法：br_add_if 和 br_del_if。

网桥转发帧时必须查询转发信息库，实现这一功能的代码定义在文件 net/bridge/br_fdb.c 中。转发信息数据库本身由 net_bridge 结构中定义的哈希表来实现。哈希表中的每一项都是 net_bridge_fdb_entry 结构的一个实例，该结构表示从网桥任意端口学习到的一个 MAC 地址信息。可以使用 fdb_find 函数在信息库中查找所需的内容。此外，该文件中还提供了添加和删除项目等信息库维护函数。

另外，为网桥提供 STP 协议支持的代码在 net/bridge 目录中以 br_stp 开头的文件中。

使用 Linux 中的网桥模块来实现软网桥时，可以使用 Netfilter/Iptables 框架来处理网

桥设备本身发送和接收的帧，但不包括网桥转发的帧。根据前面的介绍，Netfilter/Iptables框架在分组处理过程中设置钩子函数，从而使开发人员可以获得对网络协议栈中的报文进行自定义处理的机会，常见的处理包括分组过滤和 NAT 等。Linux 中通过 Ebtables(Ethernet Bridge Tables)框架在网桥转发处理过程中设置钩子函数点，来获得处理被网桥转发的数据帧的机会。Ebtables 不仅可以处理 IP 报文，还能够处理任何协议类型的帧，常用的处理包括对目的 MAC 地址的替换等。

通过对 Netfilter/Iptables 与 Ebtables 框架的结合，可以构建网桥式防火墙。这种防火墙从外部来看是一个网桥，对网络中的路由器和主机来说都是透明的，可以在不改变任何网络配置的情况下来部署它，但其却拥有防火墙的功能，能够对流经网桥的内容进行过滤和监控。

7. 流量控制管理

目前大多数路由器和交换机都支持服务质量(QoS)管理，进行流量控制，如保证某些应用获得足够的带宽，限制某些连接占用网络带宽的数量等。在 Linux 网络协议栈中既可以控制向外的流量，也可以控制向内的流量。对于向外的流量，可以采用很多不同的队列调度算法来达到各种控制目的，如保证每个连接不超过上限值，或实现类似于漏桶(leaky bucket)的策略来平滑突发的数据流量，使网络负载更为稳定。对于超过流量限制的报文可以采取延迟发送或丢弃的处理方法。但对于向内的流量只能根据一些预先设定的策略来决定是接收还是丢弃。

Linux 中的流量控制由 3 种对象来实现，它们是：Qdisc、Class 和 Filter。

(1) 排队规则

Qdisc(Queueing Discipline)是 Linux 中实现流量控制的基础。无论何时，内核如果需要通过某个网络接口发送报文，都需要按照为这个接口配置的排队规则把报文添加到输出队列。然后，内核会按照排队规则从队列中取出报文，把它们交给网络适配器进行发送。最简单的 Qdisc 是 pfifo，它不对进入队列的报文做特殊处理，只是让它们按照“先入先出”的顺序通过队列。

(2) 分类

某些 Qdisc 支持分类细化(例如 CBQ 和 HTB 等)，每个子类又可以规定自己使用的排队规则。通过分类可以组成一个 Qdisc 树，每个类都只有一个父类，而一个类可以有多个子类。

(3) 过滤器

Filter 用于为报文分类，决定它们所属的类别，进而决定它们按照何种 Qdisc 进出队列。只要报文进入具有子类的类别中，就需要借助过滤器进行分类。

图 2-5 给出了一个应用 Qdisc、Class 和 Filter 的例子。内核调用 enqueue 函数将报文交给一个支持分类的排队规则(Qdisc 1∶0)，然后通过 Filter 决定报文所属的子类别(Class 1∶1 或 Class 1∶2)，最后加入到对应的子排队规则中(Qdisc 2∶0 或 Qdisc 3∶0)。

实现流量控制模块的代码主要集中在 net/sch 目录下，其中 net/sched/sch_ *.c 文件中有各种 Qdisc 的实现方法。每种 Qdisc 的实现中最重要的是 enqueue 和 dequeue 函数。enqueue 函数会通过分类把报文加入到合适的等待队列中，而 dequeue 函数则负责从满足

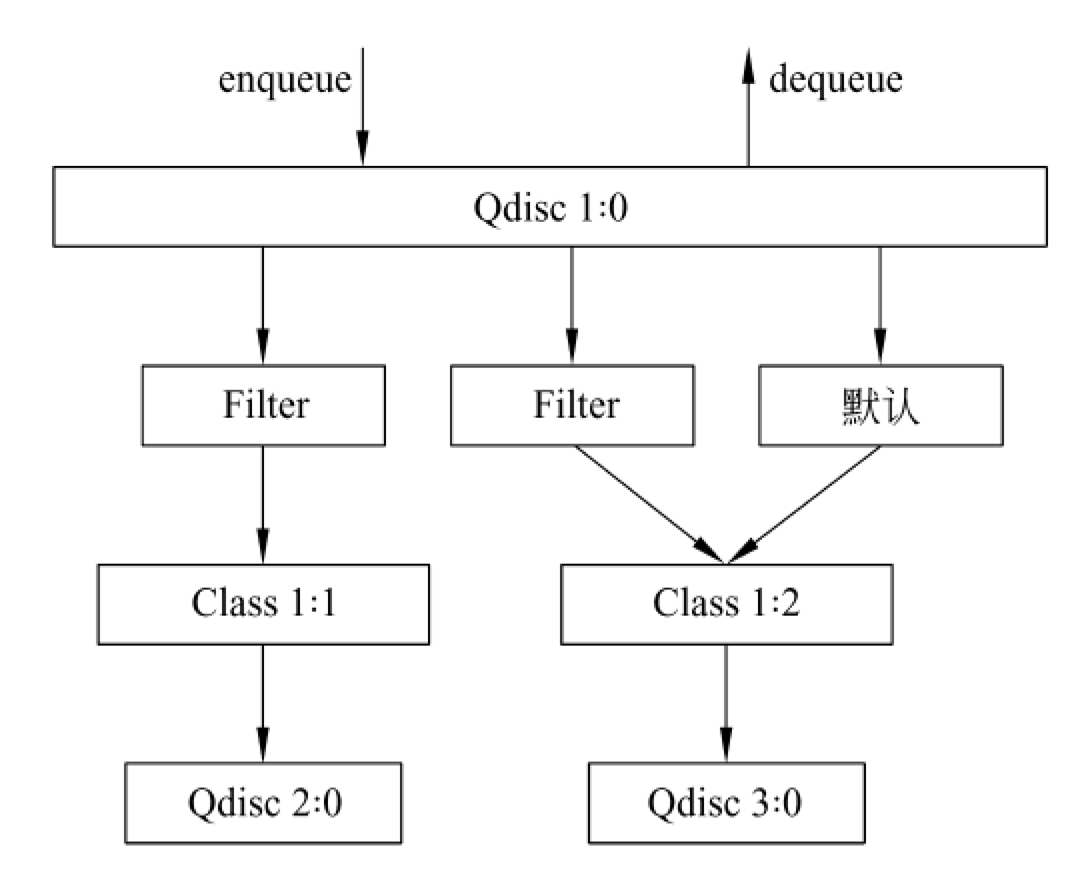


图 2-5 使用 Qdisc、Class 和 Filter 进行流量控制的示例

流量限制的各个等待队列中取出一个适当的报文进行发送，即在启用流量控制模块后，网卡所发送的全部报文都通过 dequeue 函数获取。

在文件 net/sched/sch_ingress.c 中实现了向内的流量控制，入队函数为 ingress_enqueue，如果报文超出了流量限制(quota)，则会被内核丢弃，只有在流量限制之内的报文才会被协议栈继续处理。不难看出，向内流量的 enqueue 函数实际上并没有将报文加入到任何队列之中，所以不需要实现 dequeue 函数。

在启用 QoS 功能后，用户可以通过 tc 命令来配置流量控制模块，该命令的具体使用方法可以参考 man 手册。

8. 原始套接字和 PACKET 协议族

如图 2-2 所示，在 Linux 中实现的 TCP/IP 网络协议栈源码不仅能够完成 TCP 和 UDP 协议的发送和接收流程，还支持两类比较特殊的报文处理方法：INET 协议族的原始套接字(RAW socket)和 PACKET 协议族。下面以常用的 ping 程序为例，介绍 INET 协议族的原始套接字的发送过程。

调用 ping 程序去探测其他主机时，应用程序需要在用户空间构造 ICMP 协议的 ICMP_ECHO_REQUEST 报文，然后利用 RAW socket 将数据传给内核，内核并不会处理 ICMP 报文头部的任何内容，只是为它增加合法的 IP 报头后直接发送出去，IP 层以下的发送过程并无特殊之处。目的主机收到 ICMP_ECHO_REQUEST 报文后会直接利用协议栈中的 ICMP 模块来生成 ICMP_ECHO_REPLY 报文作为回应。因此 ICMP 协议的实现代码部分在用户空间，部分在内核空间。此外，使用 RAW socket 发送报文时，还可以利用 IP_HDRINCL 选项来获得创建 IP 报头的权利，实现自定义 IP 报头的目的。

对于原始 socket 的接收流程来说，当程序中创建了一个 SOCK_RAW 类型的 socket，并指定了目标协议之后，系统开始使用 raw_v4_htable 对网卡收到的数据包进行匹配筛选。并将符合下列条件的数据包返回给应用程序：

(1) IP 报头中指定的第 4 层协议类型和 socket 指定的协议类型相同。

(2) socket 绑定了本地地址，并且与 IP 报头中的目的地址相同。

(3) socket 绑定了目的地址，并且与 IP 报头中的源地址相同。

一台主机上可以有多个 socket 同时满足这些条件，因此一个原始 IP 报文有可能同时被多个应用程序接收。所以在使用 RAW socket 时，多个应用程序可能都会收到同一个报文，每个应用程序必须自行判断是否需要该报文，接受或者丢弃这些报文。在使用 TCP 或者 UDP 协议时，完全由内核负责通过端口号将报文投递给某个特定的 socket，进而投递给拥有该特定 socket 的应用程序。即内核负责处理与 TCP 和 UDP 多路复用相关的事宜，而对于原始数据，必须由使用原始 socket 的应用程序在用户空间来进行相应的多路复用和解复用的过程。因此如果某个协议中涉及大量的多路复用和解复用操作，就应该避免使用原始 socket 来实现。

PF_PACKET 类型的 socket 是一类允许应用程序不经过网络协议栈而直接读写网络设备的接口（如图 2-2 所示）。应用程序可以针对某种报文类型创建 PF_PACKET socket，如 ADSL 拨号时所采用的 PPPoE 协议中的连接管理报文类型为 ETH_P_PPP_DISC，这意味着网络协议栈收到这类报文后，会直接投递给相应的 socket，进而被应用程序所接收并处理。另一方面，应用程序使用 PF_PACKET socket 发送的报文时，也不会经过内核网络协议栈的处理（如增加 IP 报头等），而是直接交给网络接口设备进行发送（注意：应用程序必须自己提供包括第 2 层报头在内的各级报文头部），即应用程序可以在用户态实现一个自己的网络协议栈，然后利用 PF_PACKET socket 进行发送和接收。

PF_PACKET socket 的另一种常用功能是实现嗅探器，要实现这一功能就需要 socket 能够获得各种协议类型的报文。为此 Linux 内核提供了 ETH_P_ALL 类型（如图 2-2 中所示为 ptype_all[]数组）来匹配第 2 层收到的各类报文。PF_PACKET socket 支持两个稍有不同的 socket 类型，SOCK_DGRAM 和 SOCK_RAW。前者让内核来完成添加或者删除 Ethernet 帧头部的工作，而后者则让应用程序对 Ethernet 帧头部拥有完全的控制。因为每台主机都可能收到大量报文，所以嗅探器功能十分耗费资源，这就要求程序提供一种有效的过滤机制，来屏蔽不需要收集的报文。在这方面，PF_PACKET socket 本身可以利用 bind 函数指定接收特定设备（例如，eth0）上收到的报文。另外，Linux 内核提供了一个名为 LPF（Linux Packet Filter）的过滤器，它直接应用到 PF_PACKET 接收报文的过程中，这个过滤程序可以根据使用者的定义来运行。

在编写嗅探器程序抓取所有经过网卡的报文时会使用 libpcap 库。libpcap 库在 Linux 系统中的实现依赖于 PF_PACKET socket，该库提供的是对 PF_PACKET 套接字进行适当包装后形成的一组更高级、更简单易用的接口。

通过 INET 协议族的原始套接字和 PACKET 协议族，用户程序可以自行创建或处理某些协议的报头。两者的区别主要体现在创建和接收报头的能力不同。INET 协议族的原始套接字可以创建第 4 层及其以上各层协议的报头，配合 IP_HDRINCL 选项也可以创建 IP 报头，但是网络层必须使用 IP 协议；而 PF_PACKET 协议族要求用户提供第 2 层及其以上的各级报头，因此适用于其他各种网络层协议。在接收时，INET 协议族的原始套接字只能获得 IP 报文的负载部分，而无法获得 IP 报头；而使用 PF_PACKET 协议族不但可以获得 IP 报头，还可以获得 Ethernet 帧头部的内容。

本节首先介绍了 Linux 网络子系统的设计原则，然后结合 TCP/IP 协议栈的实现简单介绍了 Linux 中常用的网络功能模块。但 Linux 网络协议栈功能繁多，本节内容不可能面面俱到，有兴趣的读者可以阅读参考资料中给出的《The Linux Networking Architecture，

Design and Implementation of Network Protocols in the Linux Kernel》和《Understanding Linux Network Internals》,这两本书来更深入地了解 Linux 网络协议栈的相关内容。

2.2 Linux 网络协议栈中报文发送和接收流程导读

本节的内容是选取 Linux 2.6.14 内核作为分析和讨论的对象,对 Linux 网络协议栈源码进行分析和总结。报文发送和接收流程的源码比较成熟,与 Linux 2.4 的内核相比变化的内容并不多。但是需要注意两个内核版本在 BSD socket 层实现上的差异,本节在分析过程中会给出相应的提示。读者可以选择一个自己熟悉的 Linux 内核版本,结合源码阅读本节的内容,对于差异之处,请读者根据本文提示的线索自行分析。

2.2.1 报文在 Linux 网络协议栈中的表示方法

要掌握网络协议栈发送和接收报文的流程,就必须首先了解内核如何表示报文。上一节介绍过 Linux 的网络协议栈在设计时就考虑到尽可能做到与具体协议无关的通用性,这种通用性也体现在报文的表示方法上。内核中使用了 socket 缓存(socket buffer)的概念来表示和管理各种协议的报文。在内核中,每一个 socket 缓存都表示一个正在被处理的报文。socket 缓存由管理区(struct sk_buff)和数据区两个部分组成,其关系如图 2-6 所示。

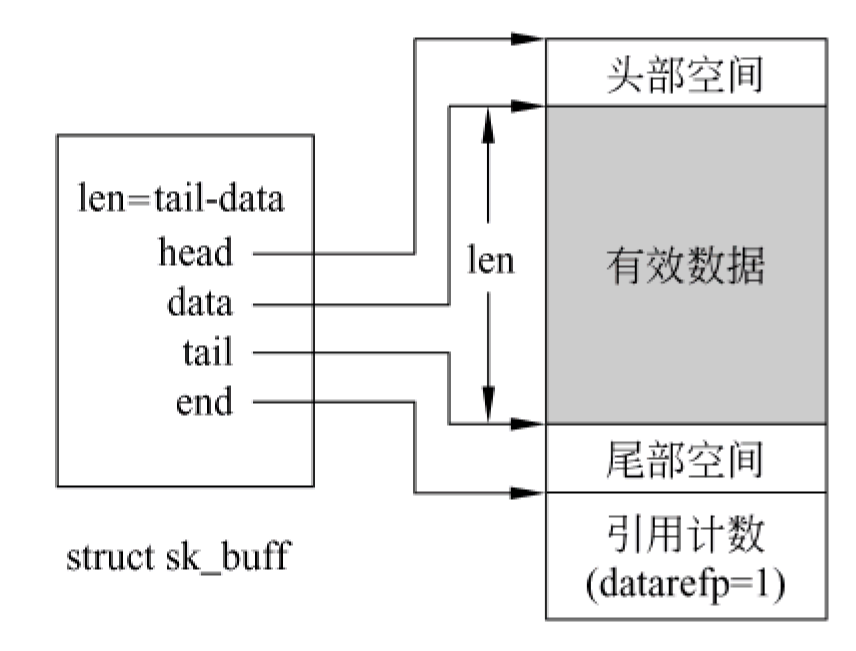


图 2-6 socket 缓存中的管理区和数据区示意图

为了通过管理区直接操作数据区,重点是要解决:如何对管理区进行有效的组织和管理。报文的管理区域由一个 sk_buff 结构来表示,通常用 skb 表示,可以认为该结构是 Linux 网络协议栈中最重要的一个数据结构,其中的字段众多,读者可以在文件＜include/linux/skbuff.h＞中找到完整的定义。

总的来说,sk_buff 中的字段可以划分为 4 类:控制内存布局的字段、协议通用字段、特定功能使用的字段和管理函数指针。

首先从两个方面对控制内存布局的字段进行讨论。

1. 控制数据区的布局

在 sk_buff 中有 4 个指针分别指向保存报文数据的内存空间中的不同位置,它们分别是 head、data、tail 和 end。这 4 个指针把数据区分成 3 个部分。

如图 2-6 所示,head 和 data 之间是头部空间,tail 和 end 之间是尾部空间,data 和 tail 之间保存真正有效的数据:报文负载和各层协议头部。除了这 4 个指针,在没有分片的情况下,字段 len 表示报文的大小,它的值一般是 tail 和 data 这两个地址的差值。网络协议栈在处理报文的过程中经常要进行添加或去除报头的操作,这就需要不断地改变报文所占用内存的大小,即要改变 socket 缓存中有效数据区域的大小。从图 2-6 可以看出,改变 data 指针的位置,可以调整头部空间的大小;改变 tail 指针的位置,则可以调整尾部空间的大小。

这两种方式都可以改变有效数据区域的大小，因而可以方便地支持处理报文头部或尾部的操作，内核中把调整 skb 指针位置的操作封装成了一组管理函数。

另一个值得注意的地方是管理区和数据区的映射关系。在内核中把表示一个报文的对应内容拆分为两个单独的内存区域来管理，以便支持在这两部分区域之间实现一对多的关系。在有多个 socket 需要接收同一个报文的情况下，例如一个是真正接收报文的应用程序打开的 socket，另一个可能是嗅探器使用的 socket，就需要为报文创建副本。两个 socket 都只对报文进行读操作，报文的数据区域可以由两个 socket 共享。因此，内核只需要创建一个管理区的副本，然后让两个 socket 通过各自的管理区结构来读取报文即可。当然，在这种方式下如果其中一个 socket 修改了报文的内容，则另一个 socket 也会受到影响。这种管理区和数据区之间的多对一关系的优点在于减少了不必要的内存分配，提高了程序运行的效率。但是，由多对一关系引出的另一问题是：当程序结束了对一个报文的处理并准备销毁其所占用的内存时，是否应该把管理区和数据区同时销毁？如果有其他管理区还在使用该数据区怎么办？解决的方法是使用引用计数。数据区域自己必须维护一个引用计数（如图 2-6 所示的 datarefp），来记录当前数据区域正被多少管理区使用。建立映射关系时，计数值加 1；销毁 sk_buff 时计数值减 1。当计数值减少到 0 时，就表示不再需要这个数据区，应该连同管理区一起销毁。有关 sk_buff 结构复制的操作将会在后面做进一步介绍。

在 sk_buff 结构中还有另一个引用计数是 users。这个字段表示 sk_buff 当前被多少个用户同时使用，即当前有多少个指针正在指向 sk_buff 所表示的管理区。与数据区的情况类似，只有当引用计数 users 的值减少到 0 时，才能释放管理区。因此，内核中可能存在多个用户使用同一个 skb，多个 skb 使用同一个数据区的情况，在销毁 skb 时也要考虑这些情况。

2. sk_buff 之间的组织方法

在内核协议栈中每个 skb 都表示一个报文，但很多报文之间具有相关性。例如，同一个 socket 中等待接收的报文，或某一个网络设备上正在等待发送的报文。Linux 会以双向链表的方式把这些相关的报文组织到一起（如图 2-7 所示）。链表的头部由结构体 sk_buff_head 表示，其中除了 next 和 prev 两个指针之外（网络协议栈通常会按照队列的方式来操作 skb 链表，因此表头的 next 指向队列的头部，prev 指向队列的尾部），还记录了链表的长度和用于访问互斥的锁。每个 skb 不仅通过 next 和 prev 指针加入到链表之中，而且都保留了指向链表头部的指针 list。除了基本的数据结构之外，Linux 的内核还提供了对该链表的操作方法，如 skb_queue_head，skb_dequeue 和 skb_insert 等。在接收报文的流程中将要介绍的 socket 的接收队列就采用了这种方式来组织报文。

除了上面介绍的两类控制内存布局的字段之外，skb 中还包含一个指向 sock 结构的指针 sk，这个 sock 结构所表示的 socket 是该 sk_buff 的所有者。如果 skb 表示本机发送或接收的报文，则应用程序以及传输层协议都必然要通过 socket 来完成收发数据的工作，此时该指针是 skb 和 socket 之间的纽带。如果 skb 表示由协议栈转发的报文，则该指针应该为空。

接下来是协议通用字段，顾名思义就是处理每个报文的过程中都需要使用到的字段，主要包括指向负责处理报文的网络设备对象的指针 struct net_device * dev 和保存报文路由信息的结构体 struct dst_entry dst，以及指向表示不同层次协议报文头部的指针结构：

```
union {…} h;
```

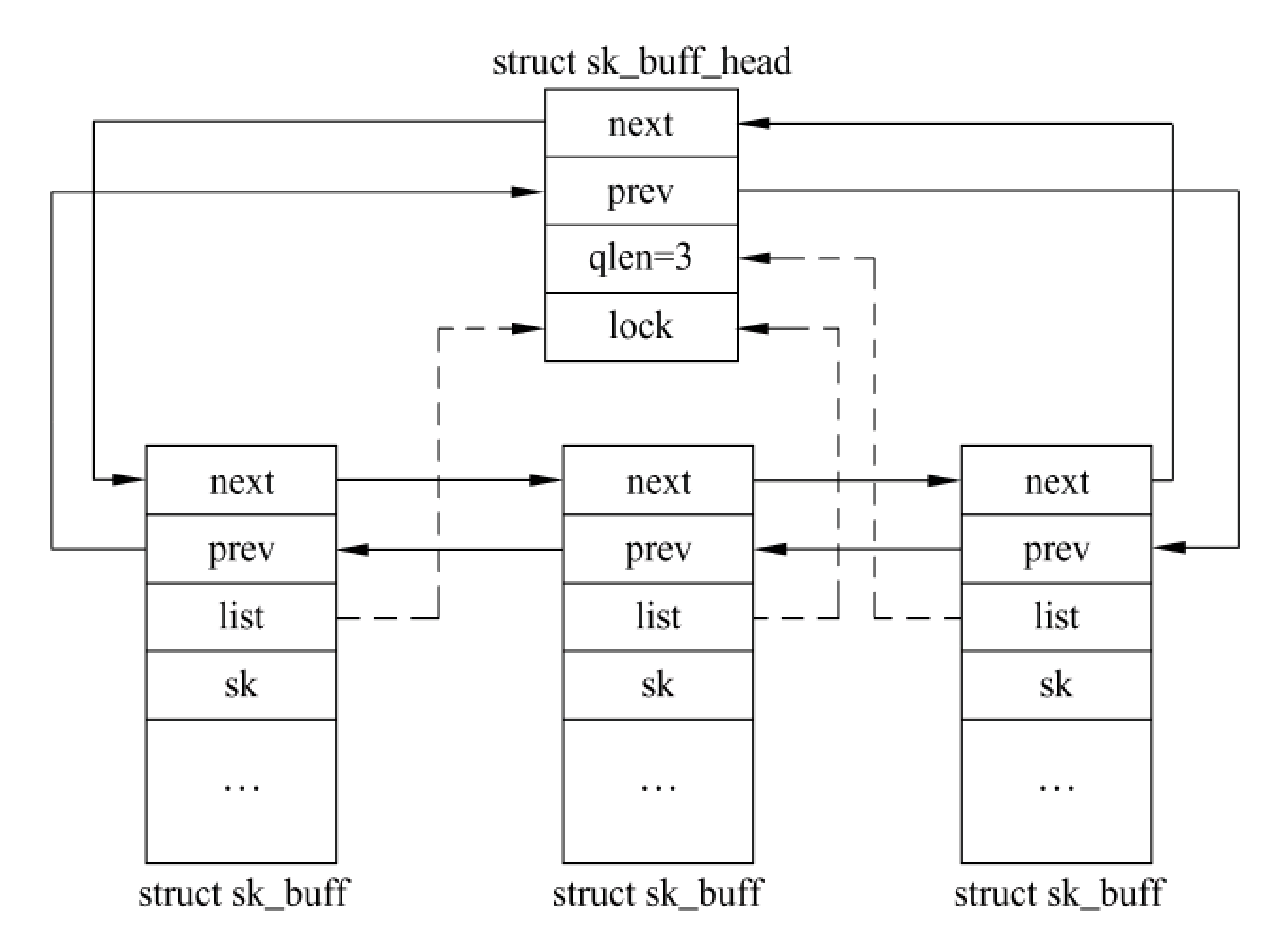


图 2-7　sk_buff 结构的双向链表示例

```
union {...} nh;
union {...} mac;
```

这 3 个指针分别指向位于 TCP/IP 协议栈中 2～4 层协议的报头：

(1) h 表示第 4 层(例如，TCP 或 UDP 报头)；

(2) nh 表示第 3 层(例如，IP 报头)；

(3) mac 表示第 2 层(例如，Ethernet 帧首部)。

每层协议的报头指针都用一个 union 结构来表示，其中罗列了所有内核支持的该层协议的头部格式。例如网络层头部的定义如下，其中包括 IPv4、IPv6 和 IPX 等协议：

```
//Network layer header
union
{
    struct iphdr      * iph;
    struct ipv6hdr    * ipv6h;
    struct arphdr     * arph;
    struct ipxhdr     * ipxh;
    unsigned char     * raw;
} nh;
```

每个 union 中都包含一个 raw 字段，它的主要作用是在还无法识别协议类型时进行初始化工作，识别出报头的协议类型之后就可以通过表示具体协议类型的头部指针来处理报头的内容了。

需要注意的是：Linux 内核从 2.6.22 开始对 sk_buff 结构进行了修改，更新了各层报头指针的处理和表示方法，即明确定义了 transport_header、network_header 和 mac_header 这 3 个指针，分别指向 2～4 层的报文头部。所以本节涉及对报文头部的处理内容皆适用于 2.6.22 之前的内核源代码，对于 2.6.22 版本之后，网络协议栈对报头处理的情况，请读者根据本文的线索自行分析。

当网络协议栈处理接收到的报文时，经过第 $n-1$ 层处理之后 skb→data 指针会指向第 n 层报头的开始位置，此时第 n 层的处理函数应该用 skb 中的第 n 层报头指针记录下第 n 层报头的地址，因为在第 n 层完成处理之后，skb→data 会移到第 $n+1$ 层报头的开始位置，如果不加以记录，则第 n 层报头的起始位置就会丢失，这个处理过程如图 2-8 所示。在发送报文的过程中，处理报头的顺序正好相反。

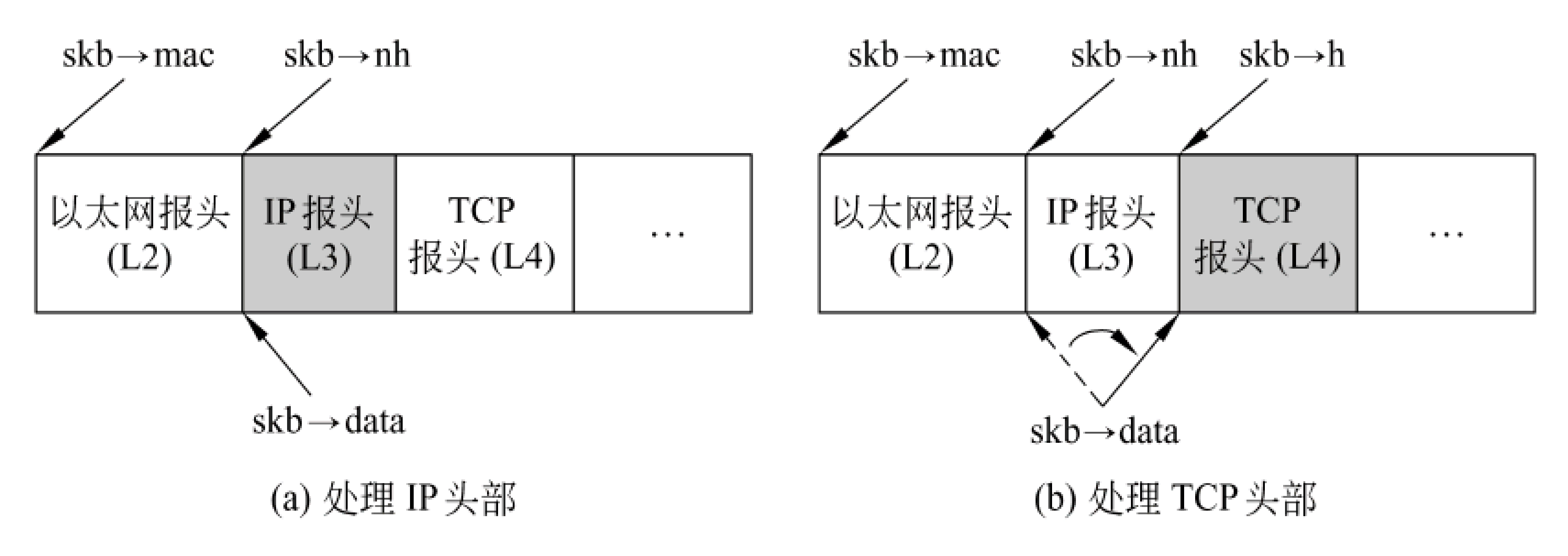


图 2-8　报文从第 3 层转发到第 4 层过程中 sk_buff 结构头部指针的变化情况示意图

因为 Linux 内核采用了模块化的设计方法，很多功能可以按需加入到内核之中，网络协议栈中的很多功能也是根据需要来决定是否提供的，例如 Netfilter。为支持这类可选的功能，sk_buff 结构中还定义了一部分特定功能使用的字段。

最后来看管理函数。内核中提供了很多用于管理 sk_buff 和 sk_buff 链表的函数，本节只对其中最重要的部分做一个简单介绍。如果读者有兴趣阅读这部分的源代码，就会发现几乎所有的函数在命名上都有两个版本：do_something 和_do_something。一般来说，后者负责完成函数名所表示的功能，前者会首先完成一些额外的检查和加锁操作，然后直接调用后者。除非在调用之前就满足条件(如加锁、更新引用计数值等)，否则不应该直接调用_do_something 版本的函数。

alloc_skb 和 kfree_skb 分别完成创建和销毁 sk_buff 的工作，新创建的 sk_buff 中 head、data 和 tail 指针全部指向数据区的开始位置，end 指针指向数据区的结束位置。在 kfree_skb 中则要处理引用计数的问题，当 skb→users 的值为 0 时，才会调用_kfree_skb 归还管理区内存，进而再调用 kfree_skbmem 来处理数据区的复用问题。

为了方便对数据区的进一步处理，内核提供了一系列操作函数，这些操作大量应用于报文的收发过程，对于阅读和理解报文收发过程的源代码具有重要意义。具体的函数是：skb_reserve(n)、skb_push(n)、skb_pull(n)、skb_put(n)和 skb_trim(n)。图 2-9 分别给出了 skb_reserve、skb_push、skb_pull、skb_put 和 skb_trim 的功能。

由图 2-9 可以看出，一旦完成对数据区内存的申请，head 和 end 指针就会固定不变。调整的目标集中在 data 和 tail 指针上，其中函数 push 和 pull 负责调整 data 指针，函数 put 和 trim 负责调整 tail 指针。上述调整函数的参数都是无符号整数，而且除了函数 skb_trim 的参数 n 表示调整之后数据区的大小之外，其他的 n 都表示调整的增量大小，因此除函数 skb_trim之外的其他函数都只能在一个方向上移动数据指针。但函数 trim 通常用于移动 tail 指针来减少数据区的大小，这也是其函数名称的由来。在 TCP/IP 协议栈的报文发送过程中，经常会使用函数 push 来增加新一层报文头部，而在报文接收处理过程中经常会使用函数 pull 来处理内层协议的头部。为了方便记忆，可以形象地认为函数 push 负责“向上

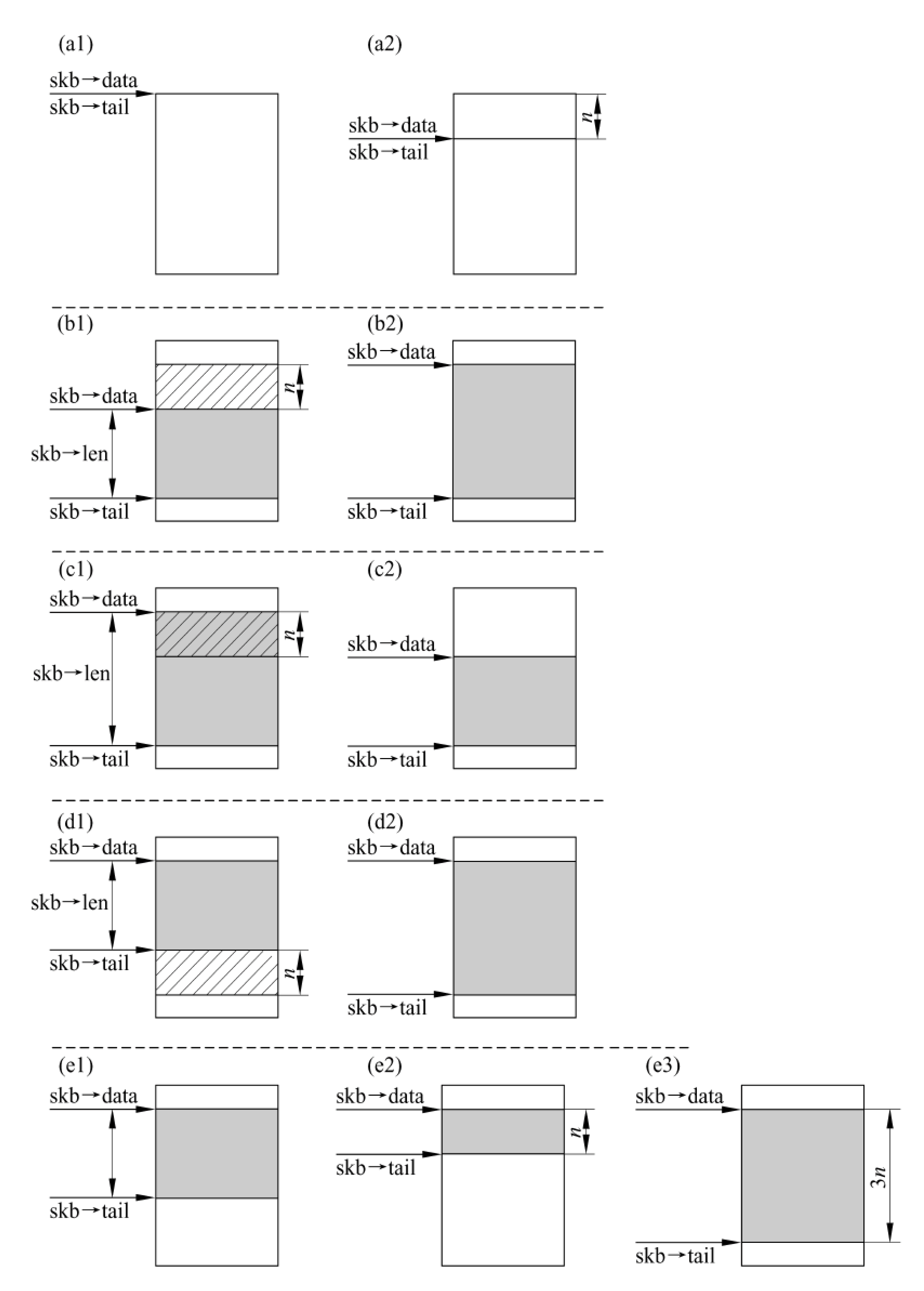


图 2-9 skb_reserve、skb_push、skb_pull、skb_put 与 skb_trim 功能示意图

推”数据指针来增加数据区的大小,函数 pull 负责“向下拉”数据指针来减少数据区的大小。

必须注意的是这些操作只会修改相应指针的位置,并不会引起数据区中数据的增加或减少,而且调用者需要负责检查是否有足够的内存缓冲区来(如头部空间和尾部空间)支持指针的移动,如果没有则返回错误。

与 sk_buff 复制操作有关的函数是:clone 和 copy。因为 skb 中包含了指向数据区的指针,所以在进行复制操作时要决定数据区是否要连同管理区一起被复制。如果是,则要选用 skb_copy 函数;如果只需要创建管理区副本,让新旧管理区共用原有的数据区,则应该选用 skb_clone 函数,此时必须更新数据区中的引用计数(如图 2-10 所示)。因为 copy 函数拷

贝的数据量更大，所以也被称为"深拷贝"，而 clone 被称为"浅拷贝"。

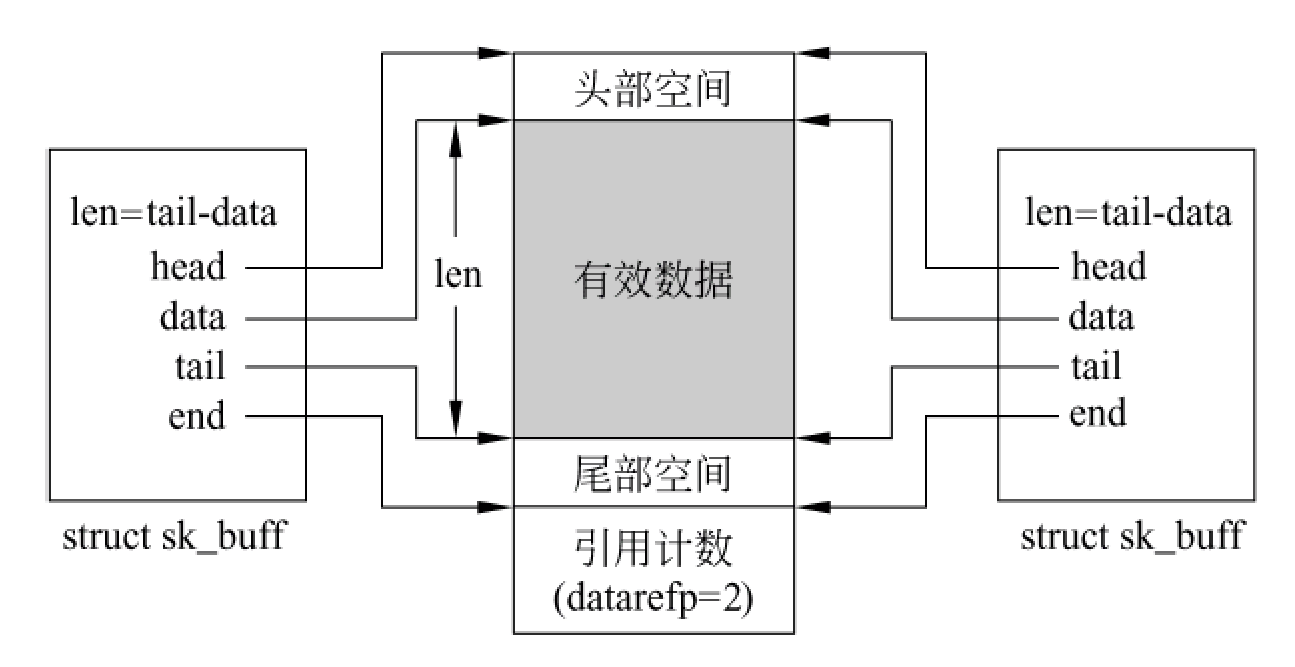


图 2-10　调用 skb_clone 后的 sk_buff 结构示意图

sk_buff 结构只是在 Linux 网络协议栈内部用来表示报文的私有数据结构，其中既包含报头也包含报文中的有效数据载荷。对于网络的用户来说(如各种网络应用程序)，只需要通过 Socket API 来完成有效数据载荷的发送和接收功能，而不需要处理报头的内容，即 Socket API 函数只需要将重点放在如何有效地在应用程序和内核中的 socket 之间传递报文中的数据载荷即可。最简单也是最常用的方法是使用一块数据缓冲区直接保存报文的内容，send 和 recv 等函数均采用这种方式。

Socket API 中还定义了一种可以实现批量数据传递的数据结构：struct msghdr，在 POSIX.1 标准中规定该结构至少需要具有如下字段：

```
struct msghdr {
    void            * msg_name;         //optional address
    socklen_t       msg_namelen;        //address size in bytes
    struct iovec   * msg_iov;           //array of I/O buffers
    int             msg_iovlen;         //number of elements in array
    void            * msg_control;      //ancillary data
    socklen_t       msg_controllen;     //number of ancillary bytes
    int             msg_flags;          //flags for received message
    ...
};
```

其中 msg_iov 指向一个由 struct iovec 组成的数组，msg_iovlen 说明数组中有效成员的个数(如图 2-11 所示)。struct iovec 中的 iov_base 会指向一块真正的数据缓冲区，而 iov_len 则表示该缓冲区的长度。这样一个 msghdr 结构总共可以表示 msg_iovlen 块有效数据区域，能够提高发送和接收 socket 函数的效率。

在 POSIX.1 标准中，使用 msghdr 结构的 Socket API 定义如下：

```
ssize_t sendmsg(int sockfd, const struct msghdr * msg, int flags);
ssize_t recvmsg(int sockfd, struct msghdr * msg, int flags);
```

不仅在应用程序中可以使用 msghdr 结构，Linux 网络协议栈的实现中也使用该结构作为发送和接收过程中表示缓冲数据的统一形式。

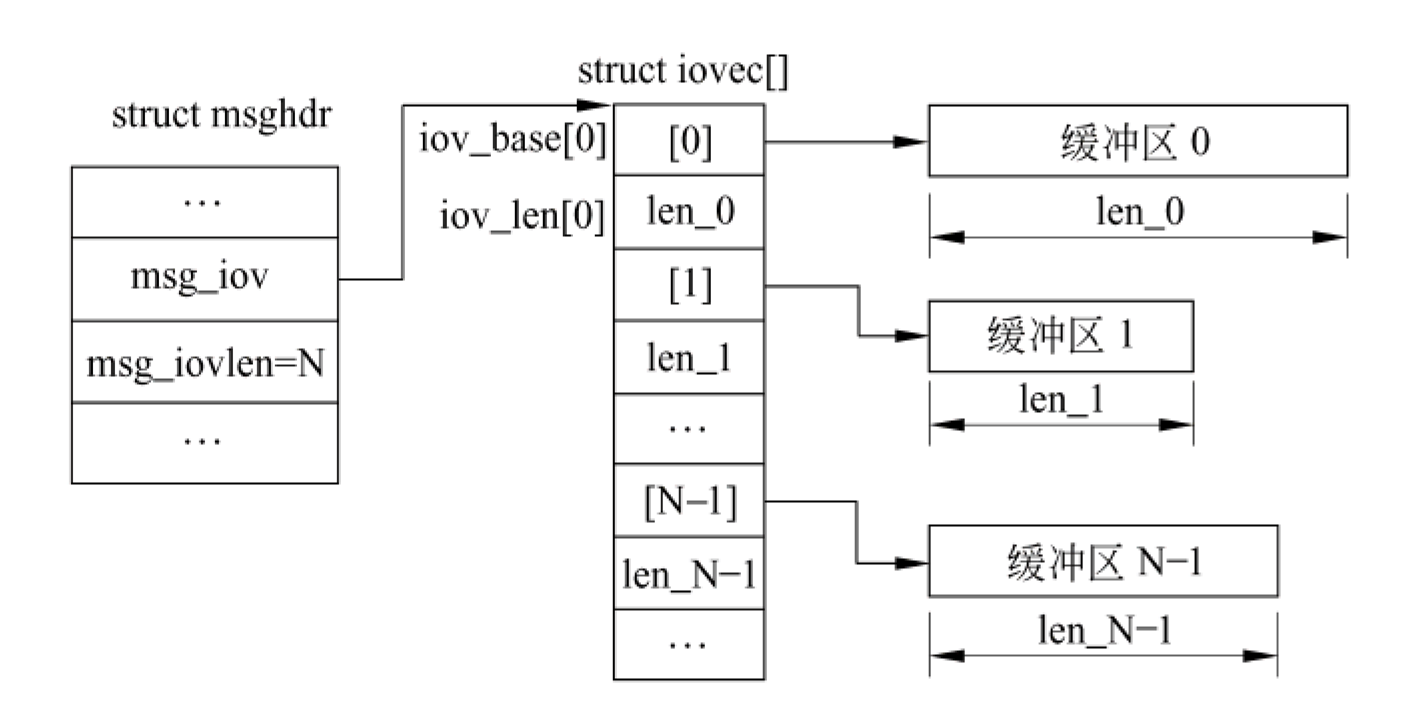


图 2-11 struct msghdr 的结构图

2.2.2 报文发送过程

下面从一个简单的基于 Client/Server 模式的 socket 网络编程实例入手，分析 Linux 网络协议栈发送和接收报文的主要流程，这里只给出了客户端程序的示意代码：

```
struct hostent * server;
int sockfd;
char buf[BUFLEN];
server=gethostbyname(SERVER_NAME);
sockfd=socket(AF_INET,SOCK_STREAM,0);                    //创建 socket
struct sockaddr_in address;
address.sin_family=AF_INET;
address.sin_port=htons(PORT_NUM);
memcpy(&address.sin_addr, server->h_addr, server->h_length);
connect(sockfd, &address, sizeof(address));             //建立连接
write(sockfd, "Hello", strlen("Hello"));                //发送数据
read(sockfd, buf, BUFLEN);                              //接收数据
close(sockfd);                                          //关闭连接
```

示例程序中报文发送的整个流程如图 2-12 所示。对于面向连接的网络应用程序，客户端在调用 connect 函数与服务器端建立连接之后，可以使用 write 函数来发送数据。对于 socket 来说可以从 4 种功能相同但接口形式不同的发送函数中任意选择一个。因为系统中使用文件描述符来表示一个已经创建的 socket，所以允许使用文件系统中通用的 write 函数来完成对数据的发送。不过 write 函数本身是为文件的写功能而创建的通用接口，无法支持 socket 通信时所特有的功能，如发送带外数据等。另外 3 个 socket 发送函数是 sendmsg、send 和 sendto，这 3 个函数都是 POSIX.1 标准中规定的 BSD Socket 必须支持的发送方法。

(1) sendmsg 函数

sendmsg 函数的原型前面已经介绍过，它的参数中使用 msghdr 结构，可以一次发送多个数据缓冲区的内容，但要求用户自行创建所需的 msghdr 结构，调用的复杂程度在这 4 种接口中最大。

(2) send 函数

send 函数与 write 函数十分相似，在保留 write 函数的三个参数的同时，增加了一个

flags 参数用来实现一些 socket 所特有的功能，例如设定 MSG_OOB 标志来发送带外数据。

（3）sendto 函数

sendto 函数接口的最大特色在于提供了设定报文目的地址和目的端口的能力。对于面向非连接的 socket 来说，只能使用 sendto 函数来进行发送。对于面向连接的 socket 来说，可以在建立连接的过程中设定目的地址和端口，因而在发送后续报文时，无需再指定地址信息。如果用户在面向连接的 socket 上调用 sendto 这类指定发送目的地址信息的接口，也可以成功发送，因为协议栈并不会使用用户指定的地址；但是反之，如果面向非连接的 socket 调用省略目的地址信息的接口则会因为缺少目的地址而出错。

从图 2-12 中可以看出，这 4 个发送接口最终都会调用_sock_sendmsg 函数（__sock_sendmsg 只是一个 inline 函数，对于 2.4 版本的内核来说，4 个发送函数最终都会执行到 sock_sendmsg 函数），而这个函数接口使用 struct msghdr 结构来表示要发送的数据。如果应用程序选择的发送函数是 sendmsg，那么只需要把位于用户态内存之中的 msghdr 结构和 struct iovec 数组拷贝到内核态即可使用。如果应用程序调用的是另外 3 种发送函数，用户态中只有一块存放待发数据的缓存，对于这些调用流程，在调用_sock_sendmsg 函数之前都可以找到创建并初始化 msghdr 结构和 struct iovec 数组的代码（具体代码在 sock_aio_write 和 sys_sendto 函数中）。需要特别注意的是在调用_sock_sendmsg 函数时，真正的发送数据还是存放在应用程序指定的用户态缓存之中。这些数据只有等到调用 tcp_sendmsg 函数准备真正发送报文时才会被拷贝到内核态。

注意在 BSD socket 层中带有 sys_前缀的函数（例如 sys_send、sys_sendmsg 等）都是 Linux 中的系统调用。在执行系统调用之后就会转入内核态进行处理，可以看出整个发送流程几乎都是在内核态中完成的。而且在这一层的处理过程中，sys_write 函数通过函数指针 file→f_op→write 调用了针对 socket 的写函数：sock_aio_write。这里函数指针的使用充分体现了代码的多态特性。函数指针的初始化代码可以在创建 socket 并为其分配文件描述符时调用的 sock_map_fd 函数中找到。

_sock_sendmsg 函数本身没有做太多的处理，只是通过函数指针 sock→ops→sendmsg 调用了 inet_sendmsg 函数，而 inet_sendmsg 函数的主要工作又是通过函数指针 sk→sk_prot→sendmsg 调用 tcp_sendmsg 函数，这两个函数指针的初始化代码都在创建 socket 时调用的 inet_create 函数中。sock→ops 指针的目的是把 BSD socket 接口映射到不同协议族的具体实现上，通过实际调用的 inet_sendmsg 函数的 inet 前缀可以看出示例程序中使用的是针对 TCP/IP 协议族的 socket 函数，其他的协议族包括 ATM、IPX/SPX 等，它们各自都有对 BSD socket 接口的实现。根据选用的具体协议族，通过 sock→ops 指针可以调用到正确的实现函数。sk→sk_prot 指针的目的同样是实现一对多的映射，只是该映射面向的是同一个协议族中不同类型的服务，例如 TCP/IP 协议族中提供的面向数据流（面向连接的 TCP 协议）、面向报文（面向非连接的 UDP 协议）和原始服务（RAW socket），从实际调用的 tcp_sendmsg 函数的前缀来看，示例程序中使用的是 TCP 协议的处理函数。

tcp_sendmsg 是上层协议发送 TCP 报文的接口，它的主要功能是分配 sk_buff 结构，并把目前还存放在用户态缓存之中的待发送数据拷贝到内核中由 sk_buff 结构指定的数据区中。生成报文的大小受到 TCP 最大段长度（MSS）的限制，因此，一次发送的数据可能需要分配到几个不同的 sk_buff 结构中，而且因为 TCP 提供的是一种数据流传输服务，不需要

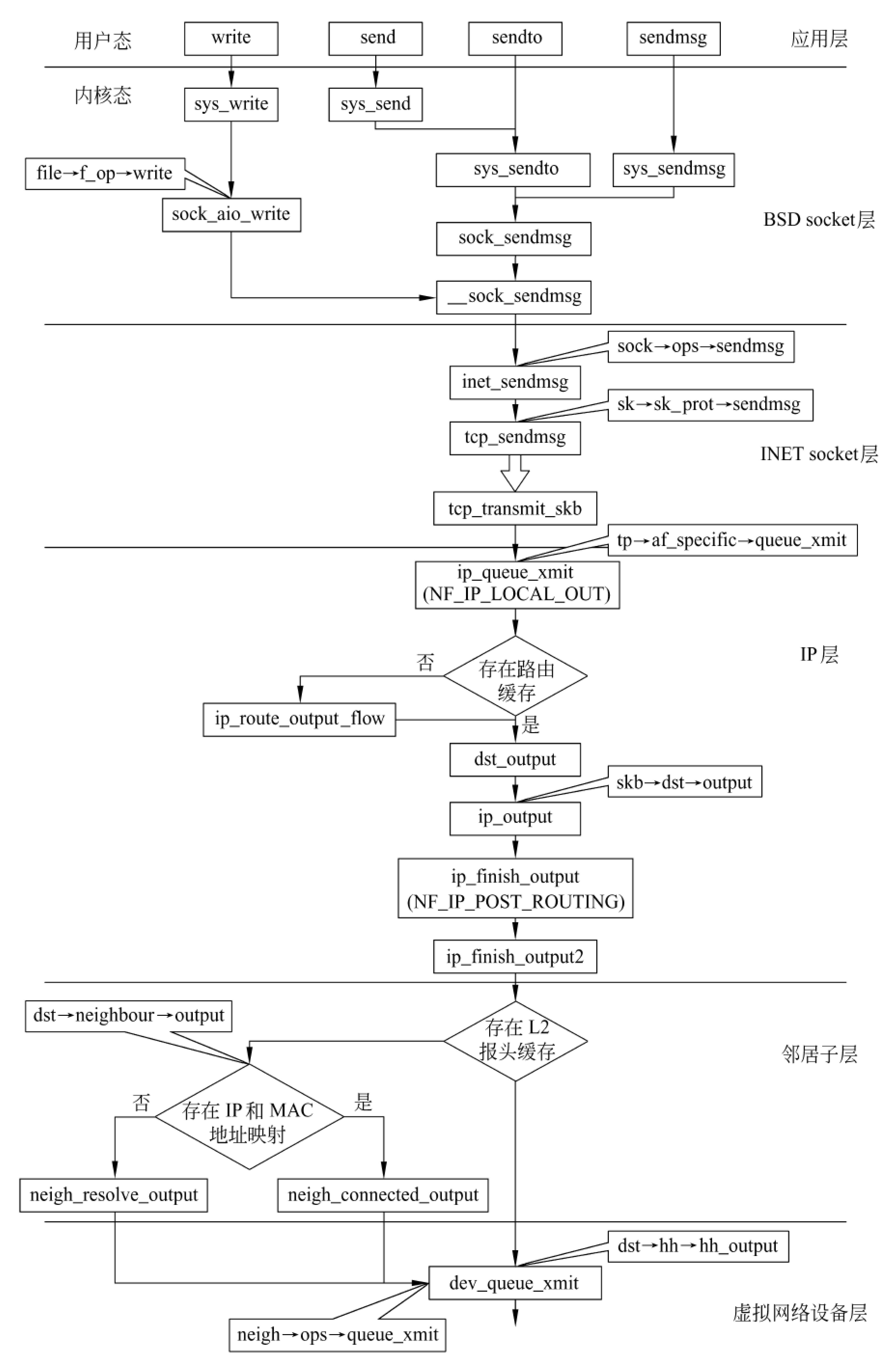


图 2-12　TCP 报文发送流程示意图

在报文中保存数据的边界，所以如果前一个 sk_buff 结构中还有剩余空间，可以先填满它，如果不足再去申请新的 sk_buff 结构。由于 TCP 协议本身在发送报文时需要考虑流量控

制、拥塞控制、重传机制以及避免糊涂窗口综合症等诸多因素，因此报文的实际发送工作在经过调度之后可能会变为异步的过程，这里不再分析该过程的实现细节，在图 2-12 中用空心箭头表示这部分被省略的调用过程。

TCP 报文的发送处理流程最终都要通过 tcp_transmit_skb 函数来完成 TCP 报头的构造，并把报文交给下一层的发送函数来进行后续的处理工作。这里需要指出的是：因为 TCP/IP 协议栈中的网络层可能是 IPv4 协议也可能是 IPv6 协议，究竟选择哪一种类型的发送函数需要根据协议栈的具体设置来决定，所以这里再次借助函数指针 tp→af_specific→queue_xmit 实现了多态特性。

如果是 IPv4 协议，实际调用的是 ip_queue_xmit 函数，该函数指针的初始化代码在创建 socket 时调用的 tcp_v4_init_sock 函数中。ip_queue_xmit 函数会为本机发送的 IP 报文查找合适的路由信息，然后创建 IP 报头并转交给下一层协议进行处理。这个函数中有以下两个地方需要说明：

(1) 关于路由缓存的使用

因为由同一个 socket 连接发送的报文具有完全相同的下一跳路由信息，所以只有连接过程中的第一个报文需要通过查找路由表来确定下一跳，其余报文可以利用路由缓存来实现选路。很明显应该为每个连接建立并维护一份路由缓存，而表示连接的 sock 结构是最适合保存路由缓存信息的位置。Linux 网络协议栈中使用 struct rtable 和 struct dst_entry 结构来表示路由缓存。sk_buff 所表示的报文完成路由的标志是其中指向 dst_entry 结构的指针 dst 指向了某一个 rtable 结构表示的路由缓存(注意 2.1.3.1 节介绍过 struct rtable 的开头包含了一个 dst_entry 结构，所以两者的指针可以进行类型转换)。sock 中用来保存路由缓存的是指向 struct dst_entry 的指针 sk_dst_cache。在 tcp_sendmsg 中创建 skb 时会检查 sock 中是否存在路由缓存，如果存在，则提前对 sk_buff 中路由缓存指针(即 skb→dst)进行初始化。ip_queue_xmit 会检查每个报文是否已经完成了路由，对于还没有下一跳信息的报文会调用函数 ip_route_output_flow 来完成选路工作，并且将路由缓存保存到该报文所属的 sock 结构中(上述工作在 sk_setup_caps 函数中调用_sk_dst_set 函数完成)，方便连接上的后续报文快速完成路由。

(2) 实现报文过滤

因为 ip_queue_xmit 函数是本机发送 TCP 报文的必经之路，所以非常适合安排对报文的检查和过滤工作，因此 Netfilter 机制选择在该函数中放置钩子函数点 NF_IP_LOCAL_OUT，用来检查从本机发送的所有报文。

ip_queue_xmit 函数最后会调用 dst_output 函数，dst_output 是一个简单的包装函数，其主要工作是通过一个无限循环来调用函数指针 skb→dst→output。有些情况下，协议栈需要在 IP 层进行一些额外的处理，如增加新的 IP 报头实现隧道功能，增加 IPSec 使用的协议头部等，这时需要先完成这些额外的处理之后才能交给下层协议来发送。为了获得更好的扩展性，Linux 网络协议栈中把这些可能的附加处理操作组织成一个函数链表，而 dst_output 函数在本质上就是遍历这个由函数指针组成的链表。链表中的最后一个函数是 ip_output，它负责把报文交给下一层进行处理。针对示例程序的发送过程，函数指针 skb→dst→output 指向的就是 ip_output 函数。

接下来在发送流程上会依次调用函数 ip_output、ip_finish_output 和 ip_finish_

output2。前两个函数都很短，易于理解，其中 ip_finish_output 函数主要是作为另一处报文过滤钩子函数的放置点。ip_finish_output2 函数则是第 3 层发送函数与邻居子系统之间的界面。因为邻居子系统使用的协议可能各不相同，所以要再次使用函数指针实现多态的效果。在 Linux 中用来表示抽象的邻居系统的结构是 struct neighbour，它定义在文件 include/net/neighbour.h 中。neighbour 结构中确实有一个函数指针集合 struct neigh_ops *ops。如果按照前面一贯的做法，应该使用下面的方法来调用发送函数：neighbor→ops→output。但实际上并非如此，因为 neigh_ops 结构中的每个函数(output、connected_output 和 hh_output)都是用于发送的函数，所以单纯使用函数指针 neighbor→ops→output 并不能取得类似于多态的效果。因此，设计者在 neighbor 结构中增加了一个函数指针 output，把它作为上层协议使用邻居系统的唯一接口函数(也是实现多态性的接口)，各种邻居协议根据情况从 neigh_ops 结构中选择一个合适的发送函数赋给 neighbor→output 指针。

邻居子系统在发送报文之前需要完成以下两项任务：

(1) 获得第 3 层地址到第 2 层地址的映射关系，对本节的例子来说就是 IP 地址与 MAC 地址的映射。

(2) 要创建好第 2 层报头。

这两项工作都可以通过使用缓存来提高处理效率。最理想的情况是缓存信息中恰好有第 2 层报头的缓存(即 hh_cache 结构中的指针 hh 不为空)，这时 ip_finish_output2 函数的处理逻辑是直接将它拷贝到 sk_buff 中来完成报头的创建，然后调用 hh_cache 结构中的函数指针 hh_output(等价于调用 dst→hh→hh_output)进行发送，它真正指向的是 dev_queue_xmit 函数，这也是把制作好的报文交给网络设备进行发送的接口函数，该函数指针在创建第 2 层报头缓存时完成初始化。

如果暂时没有合适的第 2 层报头缓存，就要调用前面介绍的邻居子系统的发送接口 neighbour→output，这个指针会根据第一项任务的完成情况指向不同的处理函数。简单来说，如果当前可以直接生成第 2 层协议的报头，即协议栈已经知道了下一跳的 IP 地址与其 MAC 地址的对应关系，此时函数指针指向 neigh_connected_output，该函数在为报文创建第 2 层报头之后，调用另一个函数指针 neigh→ops→queue_xmit(也指向 dev_queue_xmit 函数)来完成发送工作；否则就需要借助地址解析协议，如 ARP。此时 neighbour→output 指向 neigh_resolve_output 函数，该函数首先会把等待发送的报文加入到由 neighbor→arp_queue 指向的队列中，然后发送地址解析请求，并等待应答。需要指出的是，无论报文是立即发送还是在 ARP 队列中等待，函数 ip_finish_output2 都会向 IP 子系统返回成功的信号。因为此时 IP 模块已经顺利完成了对该报文的处理，后续的发送工作由邻居子系统接管。当 ARP 解析过程结束后，邻居子系统会负责将报文从等待队列中移出，交给发送设备来完成实际的发送。

从图 2-12 中可以看出，不论走哪一条路，最终都会调用 dev_queue_xmit 函数把要发送的报文交给网络接口设备。最终网络接口设备会利用驱动程序中提供的功能将报文发送到传输线路上。

最后，简单总结一下 TCP 报文发送过程中对 sk_buff 的处理情况，如图 2-13 所示。

在发送流程中，当 socket 有数据需要发送时会首先向内核申请创建 sk_buff 结构来表示报文，用于存放报文数据内容的内存应至少大于 TCP 的 MSS，这个过程的具体代码可以

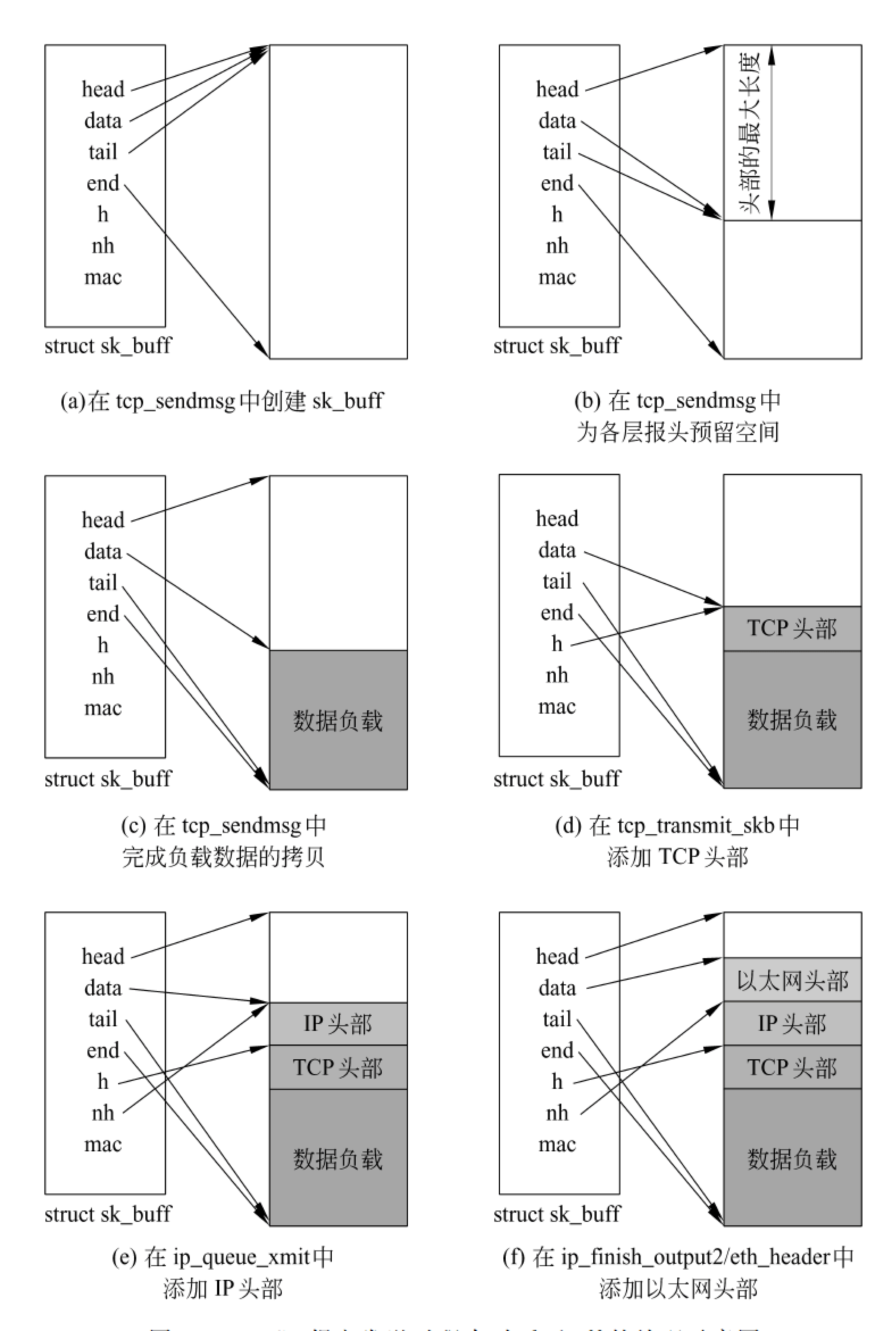


图 2-13　TCP 报文发送过程中对 sk_buff 的处理示意图

在 tcp_sendmsg 函数中找到(如图 2-13(a)所示)。在 tcp_sendmsg 函数中还会利用 skb_reserve 函数在数据区中为各层报文头(在这个例子中包括 TCP、IP 和 Ethernet 报头)预留出 MAX_TCP_HEADER 大小的空间(如图 2-13(b)所示)。这个值的大小是考虑到最坏情况下得到的结果,大多数情况下该值都会大于实际报头的大小。因为当前报文负载的大小和内容都已经确定,所以在这里还会进一步调用 skb_put 函数设定指向报文负载结尾处的指针 tail(注:具体的调用代码在 skb_add_data 函数中),并完成数据的拷贝(如图 2-13(c)所示)。接下来,在 tcp_transmit_skb 函数中会调用以下代码来根据 TCP 报头进行指针调整(如图 2-13(d)所示):

```
th= (struct tcphdr * ) skb_push(skb, tcp_header_size);
```

```
skb->h.th=th;
th->source=sk->sport;
th->dest=sk->dport;
```

再接下来，ip_queue_xmit 函数会调用以下代码来设定 IP 报头并调整 skb 中的指针情况(如图 2-13(e)所示)：

```
iph=(struct iphdr*)skb_push(skb, sizeof(struct iphdr)+(opt? opt->optlen:0));
… …
iph->saddr     =rt->rt_src;
iph->daddr     =rt->rt_dst;
skb->nh.iph    =iph;
```

最后一步处理是为报文添加 Ethernet 帧头，调整 skb 中指针的代码也会根据是否具有帧头缓存而不同。有缓存的情况下会直接在 ip_finish_output2 函数中调用 skb_push，而其他情况下都是通过函数指针 dev→hard_header 来调用 eth_header 函数最终完成对 skb_push 的间接调用(如图 2-13(f)所示)，因为后续处理不需要使用 skb→mac，所以没有为这个指针进行初始化。

2.2.3 报文接收过程

示例程序在调用 write 函数发送报文之后，紧接着调用 read 函数来接收服务器端发送的报文，接下来就深入到这个函数的内部，分析报文的接收流程。与发送流程不同，接收流程可以划分为从上到下和从下到上两个小的过程，这两部分通过 socket 中的报文接收队列衔接在一起(如图 2-14 所示)。先来讨论位于上半部分的流程。这一部分和前面介绍的报文发送流程的开始部分十分类似，从 4 个位于用户态的应用层接收函数开始，执行过程最终都汇集到_sock_recvmsg 函数，然后经过两次类似于C++中虚拟函数表(VFT)的处理，分别调用 sock_common_recvmsg 函数(对于 2.4 版本的内核来说，调用的是 inet_recvmsg 函数)和 tcp_recvmsg 函数。在 tcp_recvmsg 函数中会检查 sk→sk_receive_queue 指向的 socket 的接收队列，如果队列中有报文等待接收，则把其中的数据部分返回给用户进程(即将数据写入接收函数指定缓冲区，并使等待的接收函数成功返回)。如果此时队列中没有任何报文，则函数有两种选择，要么立即返回，要么等待数据的到来。立即返回策略能够让应用程序中调用的接收函数也即刻返回，这就是 socket 编程时介绍过的非阻塞 socket；等待策略可以使应用程序在调用接收函数时，因为没有数据可读而导致所在进程或线程进入睡眠状态，这就是阻塞的 socket(同样的情况也会发生在 accept 函数的处理过程中)。当有新的报文到达接收队列后，内核会通知(唤醒)正在等待的接收进程，让它们把收到的数据返回给对应的应用程序。

那么网络协议栈又是如何把收到的各种协议的报文分门别类地投放到对应的 socket 接收队列中的呢？这就是报文接收流程的下半部分需要解决的核心问题。现在重新把注意力移到接收报文的唯一接口：网络接口设备。以示例程序中使用的 Ethernet 为例，分组以异步的方式到达网卡，网卡收到分组之后通过中断来通知系统接收并处理收到的分组，因此分组的接收流程应该在网卡的中断处理程序中完成。从网卡收到分组开始，直到把它们投递到对应 socket 的等待队列中为止，需要经过很多处理，执行的时间比较长。而且中断处

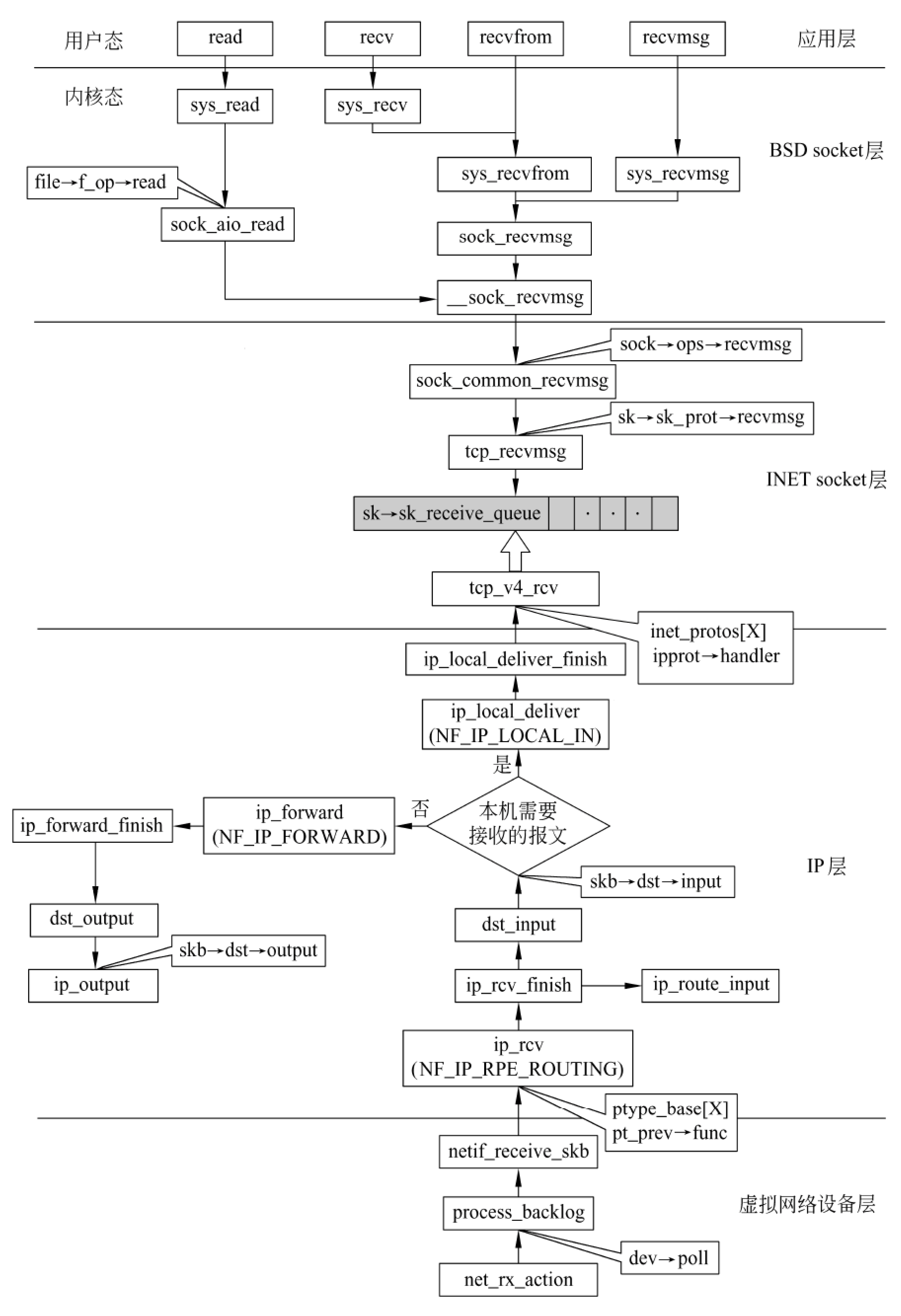


图 2-14　TCP 报文接收流程示意图

理程序为了简化程序编写的复杂性，一般会屏蔽中断嵌套。如果中断处理程序本身执行时间过长，会阻碍系统对其他中断信号的处理，从而降低系统的中断处理能力。

Linux 为了提高中断系统的效率将中断处理过程分为两部分：硬件中断也称为中断处理的上半部分(top half)和软中断也称为中断处理的下半部分(bottom half)。

硬件中断处理程序只负责与硬件有关的处理，这部分必须尽快完成，以便释放宝贵的硬

件资源，提高系统吞吐量，例如将分组从网卡上的缓冲区拷贝到内核态的缓存中等，不需要硬件参与的处理过程都交给软中断来延迟完成。需要明确的是 Ethernet 卡驱动会在硬件中断的处理过程中负责把收到的分组保存为 sk_buff 的形式，然后调用 netif_rx 函数把它交给系统的软中断部分来完成。

分组接收过程在软中断中的处理起点是 net_rx_action 函数，这也是图 2-14 表示的接收流程下半部分的起点。net_rx_action 函数中最重要的部分是通过函数指针 dev→poll 来调用某个网络设备的轮询函数。这里使用函数指针的目的是屏蔽不同设备之间的差异，提供统一的处理接口。实际上，只有最新的支持 NAPI 的网卡才支持轮询操作，其他网卡共同使用由内核提供的一个缺省处理函数：process_backlog。在函数 process_backlog 中最重要的一步是调用 netif_receive_skb 函数处理每一个已经被网卡接收，但还在等待网络协议栈做进一步处理的分组。在 netif_receive_skb 函数中才真正拉开了处理各层报头的序幕。

由于 Ethernet 的帧头部在网卡驱动中已经处理完毕，所以此时的主要任务是如何根据帧头部中上一层协议类型字段的值，把分组交给正确的上层处理函数。这是一个对号入座的过程。号码是上一层协议类型字段的值，而 Linux 内核协议栈中用一个名为 ptype_base 的全局哈希表来表示所有可能的座位(即所有支持的协议)。通过查找和比较找到表示对应座位的 packet_type 结构体的指针。找到座位之后，就会通过 packet_type 结构中的函数指针 func 来调用对应的上层协议的处理函数。在示例程序中，Ethernet 帧头部中上一层协议类型字段的值是 ETH_P_IP，表示上层是 IP 分组，则对应的处理函数是 ip_rcv。不难看出，Linux 网络协议栈中支持的每一种第 3 层协议都必须提供自己的处理函数，并把表示该协议的 packet_type 结构体(其中包括处理函数的指针)注册到全局哈希表 ptype_base 中。

这里使用函数指针也充分体现出了多态特性：同样的函数调用形式，根据报文协议的不同，而执行不同的处理函数。与发送过程的情况类似，接收过程中也应该进行一些报文的检查和过滤工作，因此 Netfilter 在 ip_rcv 中设置了一个钩子函数点，因为此时报文尚未经过路由处理，无法判断它将会被本机接收还是被继续转发，所以该过滤点也称为 PRE_ROUTING 点。经过检查点之后是 ip_rcv_finish 函数，它最主要的工作是首先调用 ip_route_input函数决定该报文是由本机接收还是需要转发，这个函数同时会设定 sk_buff 中的 dst 指针。如果是需要转发的报文，就让 skb→dst 的 input 指针指向 ip_forward 函数，并把 skb→dst 的 output 指针设置为 ip_output 函数。如果是本机将要接收的报文，就把 skb→dst 的 input 指针设置为 ip_local_deliver_finish 函数。

与发送流程中使用的 dst_output 函数类似，接下来，接收流程使用 dst_input 函数来遍历处理 skb→dst→input 指向的函数指针链表，对于转发的报文，通过 skb→dst→input 指针调用的是 ip_forward 函数，Netfilter 在该函数中设置了一个检查点来专门监控需要转发的报文，在转发报文的处理分支上会继续调用 ip_forward_finish 和 dst_output 函数，在 dst_output 函数中又会通过函数指针 skb→dst→output 调用在 ip_route_input 函数中设定的 ip_output 函数，下面的处理过程与分组发送过程基本相同。而对于本机接收的分组，则会通过 skb→dst→input 指针调用 ip_local_deliver_finish 函数。执行流程到达 ip_local_deliver_finish 函数就表示 IP 层的处理已经全部完成，此时同样也面临怎样把分组交给正确的上层协议处理函数的问题。解决策略还是对号入座。Linux 网络协议栈采用的是一个 struct inet_protocol 类型的数组 inet_protos[MAX_INET_PROTOS]表示全部候选上层协

议处理函数，inet_protocol 结构中定义了每种协议的处理函数指针 handler。具体协议的选择通过查询 IP 报头中的上一层协议字段实现。具体的调用过程仍然通过函数指针 handler 来完成。对于示例程序而言，传输层采用的是 TCP 协议，所以实际调用的处理函数是在文件 net/ipv4/af_net.c 中定义的全局变量 tcp_protocol 中设定的 tcp_v4_rcv 函数。

tcp_v4_rcv 函数是处理本机收到的 TCP 报文的起点，此时可以凭借 IP 协议和 TCP 协议报头中的信息构建四元组＜源地址、源端口、目的地址、目的端口＞，并确定该报文所属的 socket，即找出 socket 对应的 sock 结构。然后用对应的 sock 结构指针来为 sk_buff 中的 sk 字段进行赋值，并根据需要将报文对应的 sk_buff 结构添加到 sock 结构的报文接收队列中。鉴于 TCP 协议处理过程的复杂性，图 2-14 同样用一个空心箭头省略其中的处理细节。不难发现，接收流程的下半部分在本质上是一个不断解复用的过程，根据收到报文中所提供的信息，把分组转发给对应的处理函数。至此分组接收的上、下两个部分实现贯通。

在完成对 TCP 报文接收处理流程的介绍之后，再次对该过程中对 sk_buff 的处理方法进行总结，如图 2-15 所示。

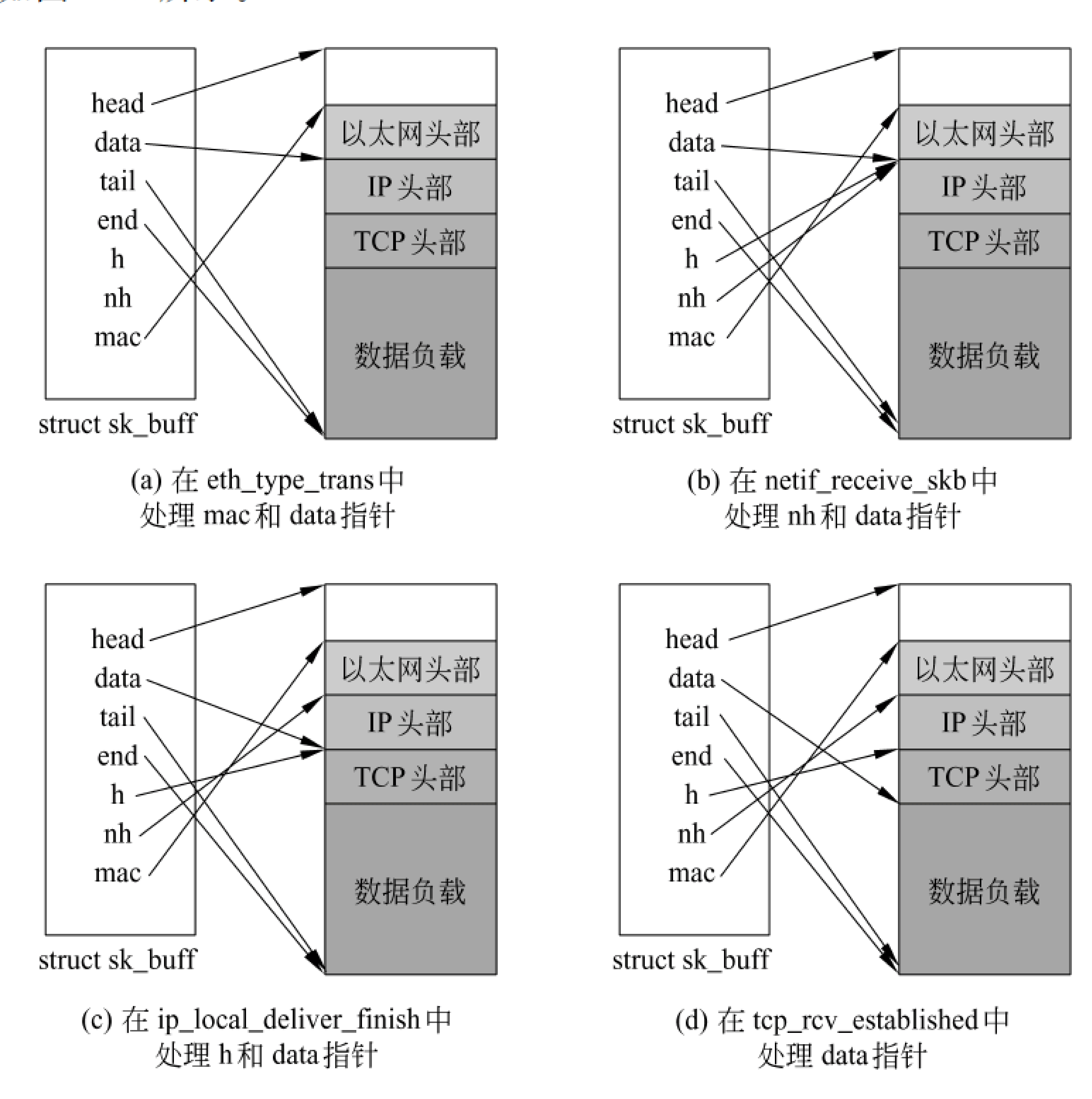


图 2-15　TCP 报文接收过程中对 sk_buff 的处理示意图

接收流程中，网络设备在收到报文后为其生成对应的 skb 结构，这一操作通常在网卡的驱动程序中完成。对于 Ethernet 卡来说，驱动程序会调用 eth_type_trans 函数完成下面两步操作：

```
skb->mac.raw=skb->data;
skb_pull(skb,dev->hard_header_len);
```

此时 skb→mac 指向了 Ethernet 的帧头，skb→data 指向 IP 报头。而在 netif_receive_skb 中，通过 skb→h. raw = skb→nh. raw=skb→data；使得 skb→nh. iphdr 指向 IP 报头。这样，网络协议栈在接下来分析和处理 IP 报头时，可以方便地使用这个指针来访问报头中的各个字段。如果处理的是发给本机的报文，则会调用 ip_local_deliver_finish 函数，并在其中执行下面两步操作：

```
__skb_pull(skb, ihl);
skb->h.raw=skb->data;
```

此时 skb→h. tcphdr 恰好指向 TCP 报头开始的地方，最后经过对 TCP 报头的处理之后在 tcp_rcv_established 函数中调用__skb_pull(skb，tcp_header_len)；这标志着网络协议栈完成了对报头的处理(应用层协议的报头除外)，其余的数据应该交给用户程序进行下一步处理。在此之后，协议栈会将 skb 加入到 sk 结构的 sk_receive_queue 队列中，并且 skb→data 指向报文的数据负载部分，在报文接收流程上半部分调用的 tcp_recvmsg 函数负责将报文中的数据负载拷贝到用户程序的缓冲区中，然后清除该 skb 结构。

理解了报文发送和接收的基本流程及其中用到的各种数据结构之后，可以很容易为协议栈添加一种新的协议，有兴趣的读者可以把这一点作为一个扩展和提高的练习。

第3章

基于DES加密的TCP聊天程序

3.1 本章训练目的与要求

DES(Data Encryption Standard)算法是一种典型的对称分组加密算法,也是应用密码学中最基本的加密算法之一,目前广泛应用于网络通信加密、数据存储加密、口令与访问控制系统之中。掌握DES算法在网络通信中的应用对于理解对称加密算法非常有益。本章以加密TCP聊天程序为任务,研究基于DES的通信加密应用软件的设计与编程方法。

本章训练的目的是:

(1) 理解对称加密算法DES的基本工作原理。

(2) 掌握将对称加密DES算法应用于网络通信的设计与软件编程的基本方法。

(3) 掌握Linux操作系统socket编程的基本方法。

本章训练的要求是:

(1) 利用socket编写一个TCP聊天程序。

(2) 通信内容经过DES加密与解密。

3.2 相关背景知识

3.2.1 DES算法的历史

随着计算机在通信中的广泛应用,社会对信息加密产品标准化的要求日益迫切。1973年,美国国家标准局NBS公开征集国家密码标准。1974年,IBM公司向NBS提交了由Tuchman博士领导的研究小组设计、改进的Lucifer算法。美国国家安全局NSA组织专家对该算法的安全性进行评估后,于1976年11月确定将其作为联邦数据加密标准,授权在非机密的政府通信中使用。DES于1977年正式成为标准,一直在美国政府、军队中广泛使用。10年之后的1988年里根政府宣布DES算法服役期满,转为民用。后来又被美国商界和世界其他国家广泛采用。NSA宣布每隔5年对DES重新审议一次,以确定其是否还适合继续作为联邦标准。1994年1月,NSA宣布DES的使用寿命延续到1998年。

1984年美国总统签署145号国家安全决策令(NSDD),命令NSA着手发展新的密码标准。2001年,高级加密标准AES取代DES成为了新的密码标准。

虽然DES已不再作为数据加密标准,但是DES算法仍然是世界很多组织采用的基本密码标准,广泛地应用于网络通信的数据加密、数据存储加密、口令与访问控制系统之中。

研究 DES 算法的设计思想与应用技术，有助于读者深入理解分组密码的设计方法，掌握分组密码在网络通信中应用的基本原理、设计与实现技术。

3.2.2 DES 算法的主要特点

DES 算法的特点可以归纳为以下几点：

(1) DES 是一种对称分组密码算法。明文分组长度为 64bit，密文分组长度也是 64bit。

(2) 加密过程要经过 16 圈迭代。初始密钥长度为 64bit，但其中有 8 bit 奇偶校验位，因此有效密钥长度是 56bit；子密钥生成算法产生 16 个 48bit 的子密钥，在 16 圈迭代中使用。

(3) 解密与加密采用相同的算法，并且所使用的密钥也相同，只是各个子密钥的使用顺序不同。

(4) DES 算法的全部细节都是公开的，其安全性完全依赖于密钥的保密。

3.2.3 DES 算法的基本内容

DES 算法包括初始置换 IP、逆初始置换 IP^{-1}、16 圈迭代以及子密钥生成算法。

1. 初始置换 IP

将 64bit 的明文重新排列，而后分成左右两块，每块 32bit，分别用 L_0 和 R_0 表示。IP 置换表如表 3-1 所示。通过对该表进行观察可以发现其中相邻两列的元素位置号数相差 8，前 32 个元素均为偶数号码，后 32 个均为奇数号码，这样的置换相当于将明文的各字节按列写出，各列经过偶采样和奇采样置换后，再对其进行逆序排列，将阵中元素按行读出以便构成置换的输出。

表 3-1 IP 置换表

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

2. 逆初始置换 IP^{-1}

在 16 圈迭代之后，将左右两端合并为 64bit，进行逆初始置换 IP^{-1}，得到输出的 64bit 密文，如表 3-2 所示。

表 3-2　逆初始置换表

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

输出的 64bit 为表中元素按行读出的结果。

IP 和 IP^{-1} 的输入与输出是已知的一一对应关系，它们的作用在于打乱原来输入的 ASCII 码顺序，并将原来明文的校验位 p_8，p_{16}，…，p_{64} 变为 IP 输出的一个字节。

3. 16 圈迭代

16 圈迭代是 DES 算法的核心部分。将经过 IP 置换后的数据分成 32bit 的左右两段，进行 16 圈迭代，每轮迭代只对右边的 32bit 进行一系列的加密变换，在一轮加密变换结束时，将左边的 32bit 与右边进行异或后得到的 32bit，作为下一轮迭代时右边的段，并将这轮迭代中的右边段未经任何加密变换时的初始值直接作为下一轮迭代时左边的段，这需要在每轮迭代开始时，先将右边段保存一个副本，以便在该轮迭代结束时，将该副本直接赋值给下一轮迭代的左边段。在每轮迭代时，右边的数据段要经过的加密运算包括选择扩展运算 E、密钥加运算、选择压缩运算 S 和置换 P，这些变换合称为 f 函数。

(1) 选择扩展运算

选择扩展运算(也称为 E 盒)的目的是将输入的右边 32bit 扩展成为 48bit 输出，其变换表如表 3-3 所示。置换结果按行输出的结果即为密钥加运算 48bit 的输入。

表 3-3　选择扩展运算变换表

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

(2) 密钥加运算

密钥加运算，是将选择扩展运算输出的 48bit 作为输入，与 48bit 的子密钥进行异或运算，异或的结果作为选择压缩运算(S 盒)的输入。

（3）选择压缩运算

选择压缩运算（S 盒）是 DES 算法中唯一的非线性部分，它是一个查表运算，共有 8 张非线性的变换表，如表 3-4 至表 3-11 所示，每张表的输入为 6bit，输出为 4bit。在查表之前，将密钥加运算的输出作为 48bit 的输入，将其分为 8 组，每组 6bit，分别进入 8 个 S 盒进行运算，得出 32bit 的输出结果作为置换运算的输入。

表 3-4　选择压缩运算变换表 1

	1	2	3	4	5	6	7	8
1-8	0xe	0x0	0x4	0xf	0xd	0x7	0x1	0x4
9-16	0x2	0xe	0xf	0x2	0xb	0xd	0xb	0xe
17-24	0x3	0xa	0xa	0x6	0x6	0xc	0xc	0xb
25-32	0x5	0x9	0x9	0x5	0x0	0x3	0x7	0x8
33-40	0x4	0xf	0x1	0xc	0xe	0x8	0x8	0x2
41-48	0xd	0x4	0x6	0x9	0x2	0x1	0xb	0x7
49-56	0xf	0x5	0xc	0xb	0x9	0x3	0x7	0xe
57-64	0x3	0xa	0xa	0x0	0x5	0x6	0x0	0xd

表 3-5　选择压缩运算变换表 2

	1	2	3	4	5	6	7	8
1-8	0xf	0x3	0x1	0xd	0x8	0x4	0xe	0x7
9-16	0x6	0xf	0xb	0x2	0x3	0x8	0x4	0xf
17-24	0x9	0xc	0x7	0x0	0x2	0x1	0xd	0xa
25-32	0xc	0x6	0x0	0x9	0x5	0xb	0xa	0x5
33-40	0x0	0xd	0xe	0x8	0x7	0xa	0xb	0x1
41-48	0xa	0x3	0x4	0xf	0xd	0x4	0x1	0x2
49-56	0x5	0xb	0x8	0x6	0xc	0x7	0x6	0xc
57-64	0x9	0x0	0x3	0x5	0x2	0xe	0xf	0x9

表 3-6　选择压缩运算变换表 3

	1	2	3	4	5	6	7	8
1-8	0xa	0xd	0x0	0x7	0x9	0x0	0xe	0x9
9-16	0x6	0x3	0x3	0x4	0xf	0x6	0x5	0xa
17-24	0x1	0x2	0xd	0x8	0xc	0x5	0x7	0xe
25-32	0xb	0xc	0x4	0xb	0x2	0xf	0x8	0x1
33-40	0xd	0x1	0x6	0xa	0x4	0xd	0x9	0x0
41-48	0x8	0x6	0xf	0x9	0x3	0x8	0x0	0x7
49-56	0xb	0x4	0x1	0xf	0x2	0xe	0xc	0x3
57-64	0x5	0xb	0xa	0x5	0xe	0x2	0x7	0xc

表 3-7　选择压缩运算变换表 4

	1	2	3	4	5	6	7	8
1-8	0x7	0xd	0xd	0x8	0xe	0xb	0x3	0x5
9-16	0x0	0x6	0x6	0xf	0x9	0x0	0xa	0x3
17-24	0x1	0x4	0x2	0x7	0x8	0x2	0x5	0xc
25-32	0xb	0x1	0xc	0xa	0x4	0xe	0xf	0x9
33-40	0xa	0x3	0x6	0xf	0x9	0x0	0x0	0x6
41-48	0xc	0xa	0xb	0xa	0x7	0xd	0xd	0x8
49-56	0xf	0x9	0x1	0x4	0x3	0x5	0xe	0xb
57-64	0x5	0xc	0x2	0x7	0x8	0x2	0x4	0xe

表 3-8　选择压缩运算变换表 5

	1	2	3	4	5	6	7	8
1-8	0x2	0xe	0xc	0xb	0x4	0x2	0x1	0xc
9-16	0x7	0x4	0xa	0x7	0xb	0xd	0x6	0x1
17-24	0x8	0x5	0x5	0x0	0x3	0xf	0xf	0xa
25-32	0xd	0x3	0x0	0x9	0xe	0x8	0x9	0x6
33-40	0x4	0xb	0x2	0x8	0x1	0xc	0xb	0x7
41-48	0xa	0x1	0xd	0xe	0x7	0x2	0x8	0xd
49-56	0xf	0x6	0x9	0xf	0xc	0x0	0x5	0x9
57-64	0x6	0xa	0x3	0x4	0x0	0x5	0xe	0x3

表 3-9　选择压缩运算变换表 6

	1	2	3	4	5	6	7	8
1-8	0xc	0xa	0x1	0xf	0xa	0x4	0xf	0x2
9-16	0x9	0x7	0x2	0xc	0x6	0x9	0x8	0x5
17-24	0x0	0x6	0xd	0x1	0x3	0xd	0x4	0xe
25-32	0xe	0x0	0x7	0xb	0x5	0x3	0xb	0x8
33-40	0x9	0x4	0xe	0x3	0xf	0x2	0x5	0xc
41-48	0x2	0x9	0x8	0x5	0xc	0xf	0x3	0xa
49-56	0x7	0xb	0x0	0xe	0x4	0x1	0xa	0x7
57-64	0x1	0x6	0xd	0x0	0xb	0x8	0x6	0xd

表 3-10 选择压缩运算变换表 7

	1	2	3	4	5	6	7	8
1-8	0x4	0xd	0xb	0x0	0x2	0xb	0xe	0x7
9-16	0xf	0x4	0x0	0x9	0x8	0x1	0xd	0xa
17-24	0x3	0xe	0xc	0x3	0x9	0x5	0x7	0xc
25-32	0x5	0x2	0xa	0xf	0x6	0x8	0x1	0x6
33-40	0x1	0x6	0x4	0xb	0xb	0xd	0xd	0x8
41-48	0xc	0x1	0x3	0x4	0x7	0xa	0xe	0x7
49-56	0xa	0x9	0xf	0x5	0x6	0x0	0x8	0xf
57-64	0x0	0xe	0x5	0x2	0x9	0x3	0x2	0xc

表 3-11 选择压缩运算变换表 8

	1	2	3	4	5	6	7	8
1-8	0xd	0x1	0x2	0xf	0x8	0xd	0x4	0x8
9-16	0x6	0xa	0xf	0x3	0xb	0x7	0x1	0x4
17-24	0xa	0xc	0x9	0x5	0x3	0x6	0xe	0xb
25-32	0x5	0x0	0x0	0xe	0xc	0x9	0x7	0x2
33-40	0x7	0x2	0xb	0x1	0x4	0xe	0x1	0x7
41-48	0x9	0x4	0xc	0xa	0xe	0x8	0x2	0xd
49-56	0x0	0xf	0x6	0xc	0xa	0x9	0xd	0x0
57-64	0xf	0x3	0x3	0x5	0x5	0x6	0x8	0xb

S 盒算法流程如下。假设输入的 48bit 为 $R_1R_2R_3\cdots R_{47}R_{48}$，需要将其转换为 32bit 值，先把输入值视为由 8 个 6bit 的二进制块组成，如下所示。

$$a=a_1a_2a_3a_4a_5a_6=R_1R_2R_3R_4R_5R_6$$
$$b=b_1b_2b_3b_4b_5b_6=R_7R_8R_9R_{10}R_{11}R_{12}$$
$$c=c_1c_2c_3c_4c_5c_6=R_{13}R_{14}R_{15}R_{16}R_{17}R_{18}$$
$$d=d_1d_2d_3d_4d_5d_6=R_{19}R_{20}R_{21}R_{22}R_{23}R_{24}$$
$$e=e_1e_2e_3e_4e_5e_6=R_{25}R_{26}R_{27}R_{28}R_{29}R_{30}$$
$$f=f_1f_2f_3f_4f_5f_6=R_{31}R_{32}R_{33}R_{34}R_{35}R_{36}$$
$$g=g_1g_2g_3g_4g_5g_6=R_{37}R_{38}R_{39}R_{40}R_{41}R_{42}$$
$$h=h_1h_2h_3h_4h_5h_6=R_{43}R_{44}R_{45}R_{46}R_{47}R_{48}$$

其中 a、b、…、h 都是 6 位，故其十进制值范围为 0～63，将转换后的十进制数值加一与对应表中的十六进制数值对应，查表得到 8 个 4bit 的结果，将其串在一起的 32bit 结果作为置换运算的输入。其中，a 对应表 3-4，b 对应表 3-5，c 对应表 3-6，d 对应表 3-7，e 对应表 3-8，f 对应表 3-9，g 对应表 3-10，h 对应表 3-11。

例如：

a. a＝31，则到表 3-4 中找到 32 的位置，把对应的结果 0x8 赋给 a。

b. d＝52，则到表 3-7 中找到 53 的位置，把对应的结果 0x3 赋给 d。

c. g＝15，则到表 3-10 中找到 16 的位置，把对应的结果 0xa 赋给 g。

这里需要说明，有些教材在数据压缩这一步所采用的方法与本章不同，但最终的结果是相同的。

(4) 置换运算

置换运算 P 是一个 32bit 的换位运算，对选择压缩运算输出的 32bit 数据按表 3-12 进行换位。将数据 $R_1R_2R_3\cdots R_{31}R_{32}$ 转换成为 $R_{16}R_7R_{20}\cdots R_4R_{25}$。

表 3-12　置换运算变换表

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

至此，最终获得的 32bit 数据，即为此轮迭代的输出。此输出与左边的 32bit 进行异或作为下一轮的右边段，进行加密运算前的原始的右边段作为下一轮的左边段。

加密过程的流程如图 3-1 所示。

其中，f(R，K)＝P(S(E(R)^K))，R 为某一轮迭代的右边段，K 为该轮密钥，E 为选择扩展运算，S 为选择压缩运算，P 为置换运算。

4. 子密钥的生成

64bit 初始密钥经过置换选择 PC-1、循环左移运算 LS、置换选择 PC-2，产生 16 圈迭代所用到的子密钥 k_i。初始密钥的第 8、16、24、32、40、48、56、64 位是奇偶校验位，其余 56 位为有效位。下面以第 *N* 轮子密钥的产生为例进行说明。

(1) 置换选择 PC-1

置换选择 PC-1 只在第一轮子密钥的产生过程中需要使用，它的目的是从 64bit 初始密钥中选出 56bit 有效位。选择的过程是一个查表过程，如表 3-13 和表 3-14 所示，输出的 56bit 被分为两组，每组 28bit，分别进入 C 寄存器(查表 3-13 的结果)和 D 寄存器(查表 3-14 的结果)中，准备进行循环左移。

表 3-13　密钥置换选择 PC-1 进入 C 寄存器

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	50	44	36

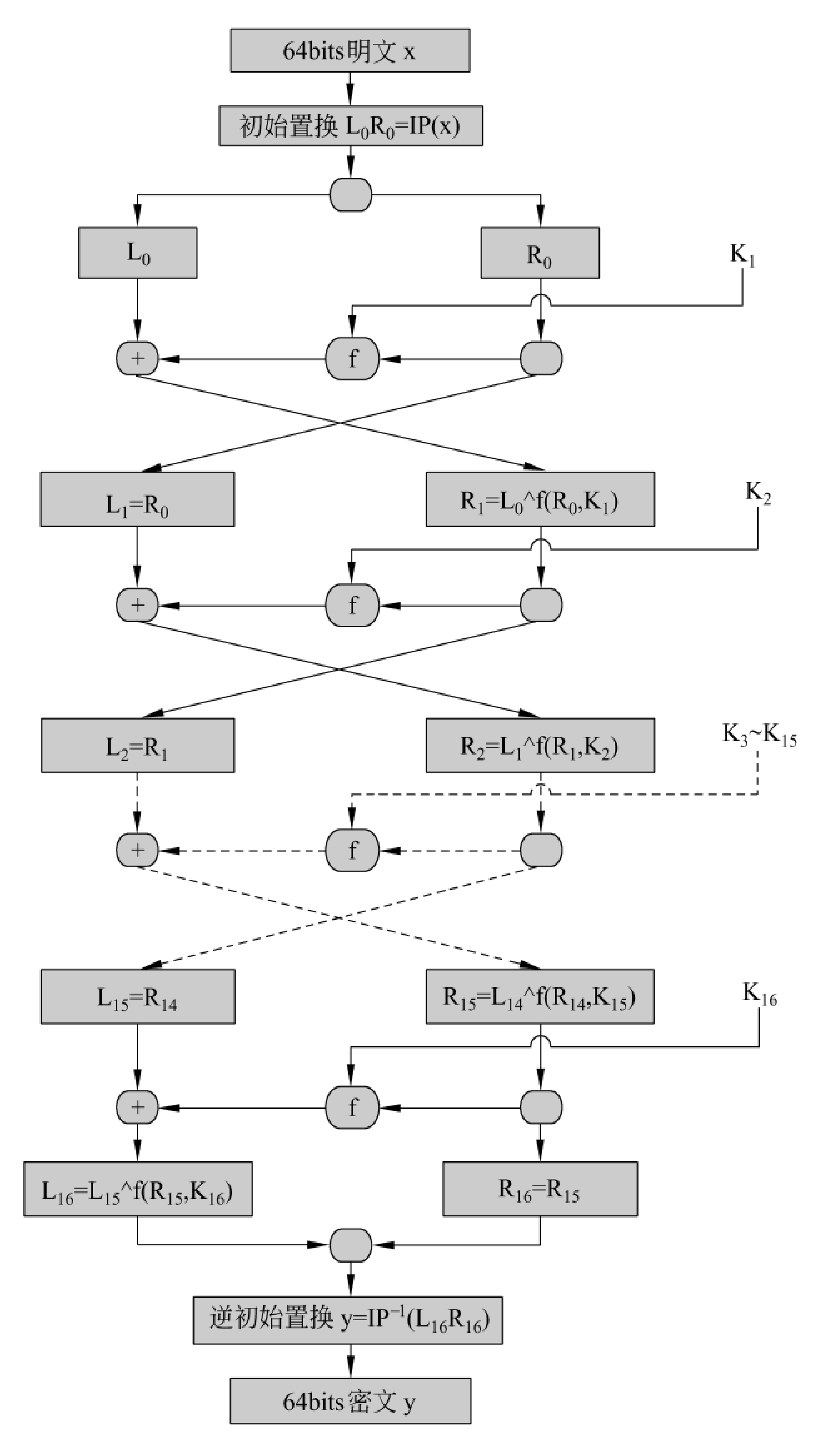


图 3-1　DES 加密流程图

表 3-14　密钥置换选择 PC-1 进入 D 寄存器

63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

(2) 循环左移 LS

此轮密钥的产生所需要循环左移的位数即为表 3-15 中的第 *N* 个元素(首元素为第一个)。C、D 寄存器中的 28bit 经过循环左移后,拼接为 56bit 作为此轮置换选择 PC-2 的输

入，同时也作为第 $N+1$ 轮子密钥循环左移的输入。

表 3-15　密钥循环左移位数表

1	1	2	2	2	2	2	2
1	2	2	2	2	2	2	1

(3) 置换选择 PC-2

置换选择 PC-2 将输入的 56bit 中的第 9、18、22、25、35、38、43、54 位删去，将其余位置按照表 3-16 置换位置，输出 48bit，作为第 N 轮的子密钥。

表 3-16　密钥置换选择 PC-2

14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

生成 16 轮子密钥的流程如图 3-2 所示。

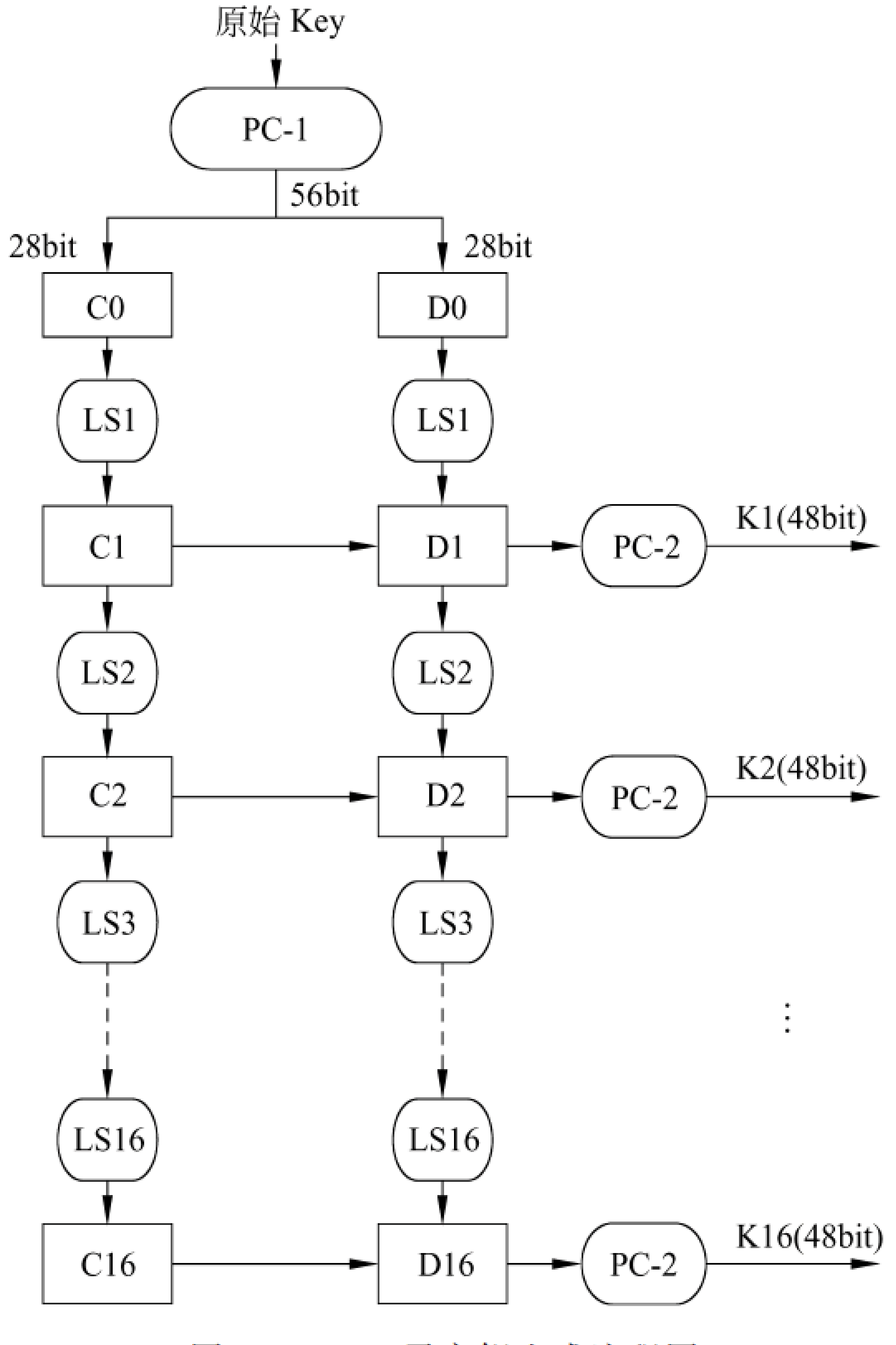


图 3-2　DES 子密钥生成流程图

3.2.4 TCP 协议

TCP 协议是一种面向连接、面向字节流的可靠传输层协议。两台采用 TCP 协议通信的计算机首先要建立 TCP 连接。TCP 协议以它自己的方式缓存数据，缓存过程对程序员和用户是透明的。TCP 协议采用“捎带确认(piggybacking ACK)”的方法，允许双方同时发送数据。TCP 规定了报文段的最大报文段长度(MSS)，默认的 MSS 值为 536 个字节。

TCP 报头的固定长度是 20 个字节，如果使用一些选项，TCP 报头的最大长度为 60 个字节。TCP 报头的结构如图 3-3 所示。TCP 报头的各字段依次为：

(1) 源端口：16 位，本地 TCP 端口号。

(2) 目的地端口：16 位，目的 TCP 端口号。

(3) 序号：32 位，用来跟踪发送报文字节顺序的序号。

(4) 确认编号：32 位，表示已经正确接收字节的序号。

(5) 报头长度：4 位，表示以 4 个字节为单位的报头长度。如果不使用选项，报头长度值为 5。

(6) 保留：6 位，留做以后使用。

(7) 标记：6 位，每一位标记都有具体的含义。

a. URG：紧急字段指针。如果为 1，此时 TCP 协议报头结构中的紧急指针有效，表示数据包中包含紧急数据。

b. ACK：确认标志位。如果为 1，表示包中的确认号有效。

c. PSH：推送功能。如果为 1，表示接收端应尽快将数据传送给应用层。

d. RST：重置位，用来复位一个连接。RST 标志置位的数据包称为复位包。一般情况下，如果 TCP 收到的一个分段明显不属于该主机上的任何一个连接，则向远端发送一个复位包。

e. SYN：用来建立连接，让连接双方同步的序列号。如果 SYN=1 且 ACK=0，表示数据包为连接请求；如果 SYN=1 且 ACK=1，则表示接受连接。

f. FIN：表示发送端已经没有数据要求传输了，请求释放连接。

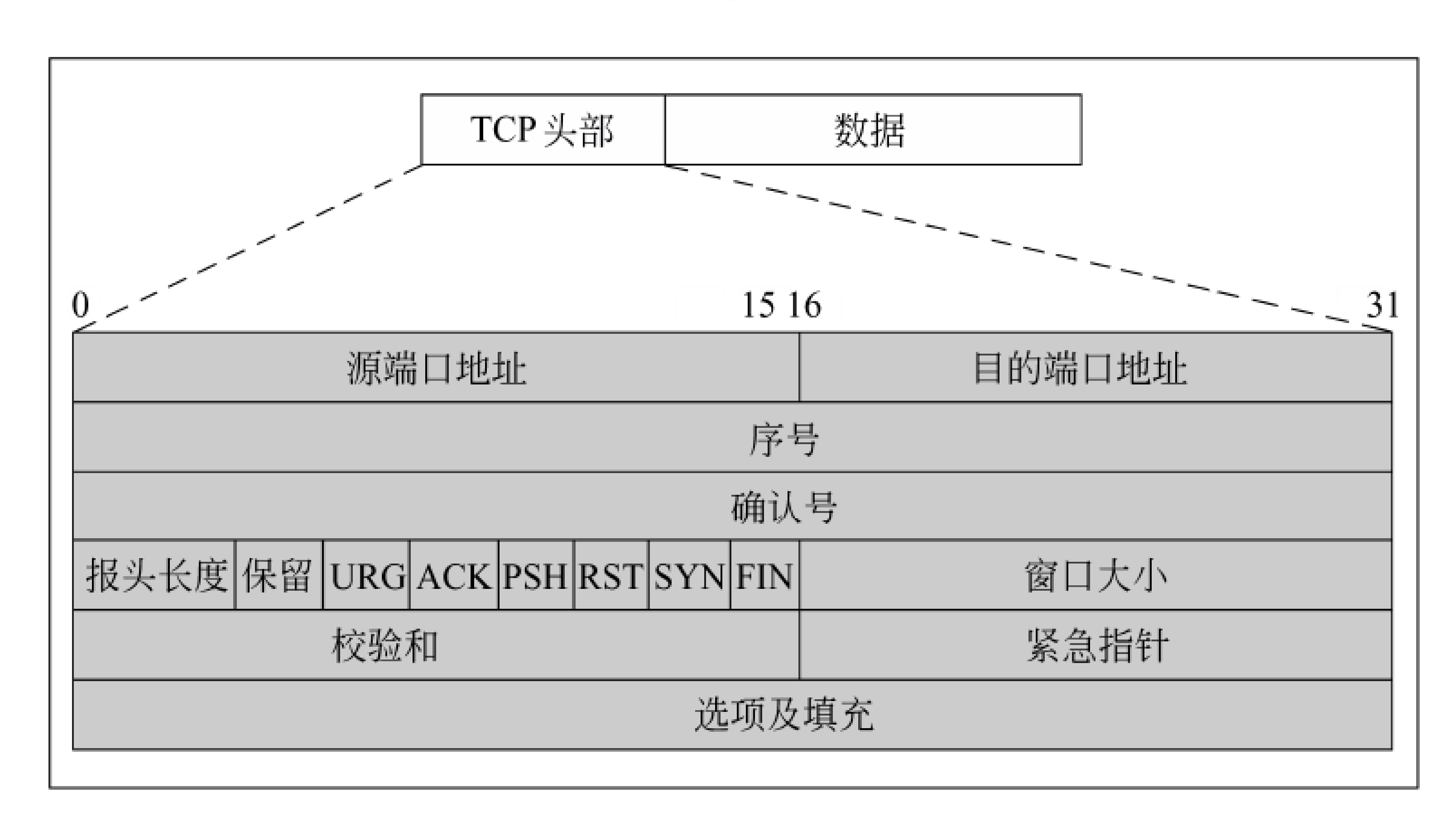


图 3-3 TCP 数据包格式

g. 窗口尺寸：16 位，表示要求对方必须维持的窗口字节数。

h. 校验和：16 位，TCP 报头和数据的校验和。

i. 应急指针：16 位，指向跟在 URG 数据后面的数据的序列号的偏移值。

j. 选项：MSS、窗口比例等。

k. TCP 连接的两端使用两对 IP 地址和端口识别这个连接，并且向监听这个端口的应用程序发送数据。

3.2.5　套接字

套接字(socket)是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。它把复杂的 TCP/IP 协议族隐藏在 socket 接口的背后，通过调用简单的 socket 函数完成特定协议的数据传输任务。TCP/IP 提供了 3 种类型的套接字：

(1) 流式套接字(SOCK_STREAM)

流式套接字是面向连接的、可靠的数据传输服务，可保证无差错、无重复且按发送顺序提交给接收方。流式套接字在传输层使用 TCP 协议。

(2) 数据报套接字(SOCK_DGRAM)

数据报套接字提供无连接服务，数据以独立的数据报形式被传送，并且在传输过程中没有差错控制和流量控制，数据可能丢失、重复或者乱序。数据报套接字在传输层使用 UDP 协议。

(3) 原始套接字(SOCK_RAW)

原始套接字允许对较低层协议(如网络层的 IP、ICMP)直接进行访问。用于实现自己定制的协议或对数据报作较低层的控制。

在本章的编程中使用的是流式套接字。

3.2.6　TCP 通信相关函数介绍

Linux 系统通过套接字(socket)来进行网络编程。网络程序通过 socket 和其他几个函数的调用，会返回一个通信的文件描述符，程序员可以将这个描述符看成普通的文件描述符来操作，通过对描述符的读写操作可以实现网络中计算机之间的数据传输。这充分体现出 Linux 操作系统的设备无关性的优点。

下面介绍本章编程训练将用到的几个 Linux 常用的套接字函数。

1. socket 函数

```
int socket(int domain, int type, int protocol);
```

该函数用于创建通信的套接字，并返回该套接字的文件描述符。

参数说明：

(1) domain：说明网络程序所在的主机采用的通信协议(AF_UNIX 和 AF_INET 等)。AF_UNIX 只能够用于单一的 UNIX 系统进程间通信，而 AF_INET 针对的是 Internet，因而可以允许在远程主机之间实现通信。

(2) type：用于指定套接字的类型。

(3) protocol：用于指定套接字所使用的通信协议。正常情况下，对于给定的协议族，

只有单一的协议支持特定的套接字类型，所以一般将 protocol 参数设置为 0。

2. bind 函数

```
int bind(int sockfd, const struct sockaddr * my_addr, socklen_t addrlen);
```

该函数用于将套接字与指定端口的相连。

参数说明：

（1）sockfd：调用 socket 函数后返回的文件描述符。

（2）addrlen：sockaddr 结构的长度。

（3）my_addr：一个指向 sockaddr 结构体的指针。

sockaddr 结构体的定义如下。

```
struct sockaddr{
        unsigned short   sa_family;
        char             sa_data[14];
};
```

不过由于系统的兼容性，一般不用这个头文件，而使用另外一个结构（struct sockaddr_in）来代替。sockaddr_in 的定义如下。

```
struct sockaddr_in{
        unsigned short         sin_family;
        unsigned short int     sin_port;
        struct in_addr         sin_addr;
        unsigned char          sin_zero[8];
}
```

这里主要使用 Internet，所以 sin_family 一般为 AF_INET，sin_addr 设置为 INADDR_ANY 表示可以和任何主机通信，sin_port 是需要监听的端口号，sin_zero[8]是用来填充的。bind 函数将本地的端口同 socket 返回的文件描述符捆绑在一起。

此函数成功返回 0，失败返回－1，并设置 errno 变量。

3. listen 函数

```
int listen(int sockfd, int backlog);
```

该函数用于实现服务器等待客户端请求的功能。

参数说明：

（1）sockfd：经过 bind 操作的文件描述符。

（2）backlog：设置请求队列的最大长度。

此函数成功返回 0，失败返回－1，并设置 errno 变量。

4. accept 函数

```
int accept(int sockfd, struct sockaddr * addr, socklen_t * addrlen);
```

该函数用于处于监听状态的服务器，在获得客户机连接请求后，会将其放置在等待队列

中，当系统空闲时，服务器用该函数接受客户机的连接请求。

参数说明：

(1) sockfd：经过 listen 操作后的文件描述符。

(2) addr：指向客户端结构体 sockaddr 的指针。

(3) addrlen：addr 参数指向的内存空间的长度。

此函数调用时，服务器端的程序会一直阻塞到有一个客户程序发出连接请求为止。

此函数成功时返回最后的服务器端的文件描述符，此时服务器端就可以向该描述符写信息了；失败时返回－1，并设置 errno 变量。

5. connect 函数

```
int connect(int sockfd, const struct sockaddr * serv_addr, socklen_t addrlen);
```

该函数用于客户端向服务器发出连接请求。

参数说明：

(1) sockfd：客户端 socket 返回的文件描述符。

(2) serv_addr：存储服务器端的连接信息。

(3) addrlen：serv_addr 的长度。

此函数成功返回 0，失败返回－1，并设置 errno 变量。

6. write 函数

```
ssize_t write(int fd, const void * buf, size_t nbytes);
```

该函数用于服务器和客户端建立连接后，将 buf 中 nbytes 字节的内容写入文件描述符。

参数说明：

(1) fd：socket 返回的文件描述符。

(2) buf：指向要进行传输内容的指针。

(3) nbytes：要传输的内容的大小。

此函数成功时返回写入的字节数，失败时返回－1，并设置 errno 变量。

7. read 函数

```
ssize_t read(int fd, void * buf, size_t nbyte);
```

该函数用于从文件描述符 fd 中读取内容。

参数说明：同 write 函数。

此函数成功时返回读出的字节数，失败时返回－1，并设置 errno 变量。

8. send 函数

```
ssize_t send(int s, const void * buf, size_t len, int flags);
```

该函数的作用基本同 write 函数相同，用于将信息发送到指定的套接字文件描述符中，其功能比 write 函数更为全面。

参数说明：

(1) s：要发送信息的文件描述符。

(2) buf：指向要发送内容的指针。

(3) len：要发送数据的长度。

(4) flags：设为 0 时，其功能与 write 函数相同。其他功能由于本章程序中不会用到，故在此不做详细介绍。

此函数成功时，返回本次调用实际发送的字节数，失败时返回－1，并设置 errno 变量。

9. recv 函数

```
ssize_t recv(int s, void* buf, size_t len, int flags);
```

该函数的作用基本同 read 函数相同，用于从指定的套接字中获取信息。

参数说明：

(1) s：要读取内容的套接字文件描述符。

(2) buf：指向要保存数据缓冲区的指针。

(3) len：该缓存的最大长度。

(4) flags：同 send 函数。

此函数成功时返回 0，失败时返回－1，并设置 errno 变量。

10. close 函数

该函数用于关闭套接字，其调用形式为：close(sockfd)。

3.3 实例编程练习

3.3.1 编程练习要求

在 Linux 平台上实现基于 DES 加密的 TCP 通信。具体要求如下：

(1) 在了解 DES 算法原理的基础上，编程实现对字符串的 DES 加密、解密的操作。

(2) 在了解 Linux 操作系统中 TCP 的 socket 工作原理的基础上，编程实现简单的 TCP 通信。为简化编程细节，不要求实现一对多通信。

(3) 将上述两部分结合到一起，编程实现通信内容通过 DES 加密的 TCP 聊天程序，要求双方事先约定密钥。在发送方通过该密钥加密，然后由接收方解密，保证在网络上传输信息的保密性。

下面给出示例程序的执行流程。

程序为命令行程序，命令行格式如下：

服务器：./chat→输入 s

客户端：./chat→输入 c

程序执行过程如下(例如服务器 IP 地址为 192.168.1.91)：

```
[root@ localhost share]# ./chat
Client or Server?
```

```
s
Listening...
```

然后打开另一个控制台终端，运行客户端的可执行程序，客户端 IP 地址为 192.168.1.232(代码如下)。

```
[root@ localhost share]# ./chat
Client or Server?
c
Please input the server address:
192.168.1.91
Connect Success!
Begin to chat...
Hello~ I'm client~
```

此时，客户端已经成功连接服务器，可以开始聊天，输入"Hello～I'm client～"，服务器端发出相应响应(代码如下)。

```
[root@ localhost share]# ./chat
Client or Server?
s
Listening...
server: got connection from 192.168.1.232, port 53558, socket 4
Receive message form <192.168.1.232> : Hello~ I'm client~
Hello~  I'm server~
```

服务器端收到连接与信息之后，给客户端回复"Hello～I'm server～"，客户端也得到信息并响应(代码如下)。

```
[root@ localhost share]# ./chat
Client or Server?
c
Please input the server address:
192.168.1.91
Connect Success!
Begin to chat...
Hello~ I'm client~
Receive message form <192.168.1.91> : Hello~ I'm server~
```

聊天过程中，任何一方输入"quit"，都可以关闭此连接，结束聊天。

3.3.2 编程训练设计与分析

程序分为两个部分，DES 算法部分和 TCP 通信部分。其中，DES 算法部分是核心部分，该部分用来实现对通过 TCP Socket 传输的数据进行加密和解密的操作。

1. DES 加密解密设计实现分析

(1) DES 类的定义

在 DES 部分的实现代码中，首先定义封装 DES 操作的类 CDesOperate，类的私有成员

包括生成的 16 圈迭代密钥、初始密钥以及加密、解密流程中用到的 4 个函数。公有成员包括构造函数、析构函数以及根据上述 4 个函数封装的加密函数与解密函数，以方便调用（代码如下）。

```
typedef int INT32;
class CDesOperate
{
private:
    ULONG32 m_arrOutKey[16][2];
    ULONG32 m_arrBufKey[2];
    INT32 HandleData(ULONG32 * left , ULONG8 choice);
    INT32 MakeData(ULONG32  * left ,ULONG32  * right ,ULONG32 number);
    INT32 MakeKey(ULONG32 * keyleft,ULONG32 * keyright ,ULONG32 number);
    INT32 MakeFirstKey(ULONG32 * keyP);
public:
    CDesOperate();
    ~CDesOperate()
    INT32 Encry(char * pPlaintext,  int nPlaintextLength,  char * pCipherBuffer,
              int &nCipherBufferLength,  char * pKey,int nKeyLength);
    INT32 Decry(char * pCipher,  int nCipherBufferLength,  char * pPlaintextBuffer,
              int &nPlaintextBufferLength,  char * pKey,int nKeyLength)
```

其中 HandleData 用来执行一次完整的加密或解密操作，MakeData 用来实现 16 轮加密或解密迭代中的每一轮除去初始置换和逆初始置换的中间操作，MakeFirstKey 用来利用用户输入的初始密钥，来形成 16 个迭代用到的子密钥，MakeKey 用来形成 16 个密钥中的每一个子密钥，Encry 用来对某一段字符加密，Decry 用来对某一段密文解密。

（2）DES 算法中用到的静态数组

初始置换 IP（代码如下）：

```
static ULONG8  pc_first[64]={
                58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
                62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
                57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
                61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7
};
```

逆初始置换 IP^{-1}（代码如下）：

```
static ULONG8 pc_last[64]={
      40,8,48,16,56,24,64,32, 39,7,47,15,55,23,63,31,
      38,6,46,14,54,22,62,30, 37,5,45,13,53,21,61,29,
      36,4,44,12,52,20,60,28, 35,3,43,11,51,19,59,27,
      34,2,42,10,50,18,58,26, 33,1,41,9,49,17,57,25
};
```

按位取值或赋值（代码如下）：

```
static ULONG32  pc_by_bit[64]= {
        0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,
        0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,
        0x200000L, 0x100000L,  0x80000L, 0x40000L, 0x20000L,0x10000L,
        0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,
        0x100L, 0x80L,0x40L,0x20L, 0x10L, 0x8L, 0x4L, 0x2L, 0x1L,
        0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,
        0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,
        0x200000L, 0x100000L,  0x80000L, 0x40000L, 0x20000L, 0x10000L,
        0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,
        0x100L, 0x80L, 0x40L,0x20L, 0x10L, 0x8L,  0x4L, 0x2L, 0x1L,
};
```

置换运算 P(代码如下):

```
static ULONG8 des_P[32]= {
                    16,7,20,21, 29,12,28,17, 1,15,23,26,
                    5,18,31,10, 2,8,24,14, 32,27,3,9,
                    19,13,30,6, 22,11,4,25
};
```

选择扩展运算 E 盒(代码如下):

```
static ULONG8 des_E[48]= {
                         32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,
                         12,13,14,15,16,17,16,17,18,19,20,21,
                         20,21,22,23,24,25,24,25,26,27,28,29,
                         28,29,30,31,32,1
};
```

选择压缩运算 S 盒(代码如下):

```
static ULONG8 des_S[8][64]=
{
      {
            0xe,0x0,0x4,0xf,0xd,0x7,0x1,0x4,0x2,0xe,0xf,0x2,0xb,
            0xd,0x8,0x1,0x3,0xa,0xa,0x6,0x6,0xc,0xc,0xb,0x5,0x9,
            0x9,0x5,0x0,0x3,0x7,0x8,0x4,0xf,0x1,0xc,0xe,0x8,0x8,
            0x2,0xd,0x4,0x6,0x9,0x2,0x1,0xb,0x7,0xf,0x5,0xc,0xb,
            0x9,0x3,0x7,0xe,0x3,0xa,0xa,0x0,0x5,0x6,0x0,0xd
      },
      {
            0xf,0x3,0x1,0xd,0x8,0x4,0xe,0x7,0x6,0xf,0xb,0x2,0x3,
            0x8,0x4,0xf,0x9,0xc,0x7,0x0,0x2,0x1,0xd,0xa,0xc,0x6,
            0x0,0x9,0x5,0xb,0xa,0x5,0x0,0xd,0xe,0x8,0x7,0xa,0xb,
            0x1,0xa,0x3,0x4,0xf,0xd,0x4,0x1,0x2,0x5,0xb,0x8,0x6,
            0xc,0x7,0x6,0xc,0x9,0x0,0x3,0x5,0x2,0xe,0xf,0x9
      },
```

```
    {
        0xa,0xd,0x0,0x7,0x9,0x0,0xe,0x9,0x6,0x3,0x3,0x4,0xf,
        0x6,0x5,0xa,0x1,0x2,0xd,0x8,0xc,0x5,0x7,0xe,0xb,0xc,
        0x4,0xb,0x2,0xf,0x8,0x1,0xd,0x1,0x6,0xa,0x4,0xd,0x9,
        0x0,0x8,0x6,0xf,0x9,0x3,0x8,0x0,0x7,0xb,0x4,0x1,0xf,
        0x2,0xe,0xc,0x3,0x5,0xb,0xa,0x5,0xe,0x2,0x7,0xc
    },
    {
        0x7,0xd,0xd,0x8,0xe,0xb,0x3,0x5,0x0,0x6,0x6,0xf,0x9,
        0x0,0xa,0x3,0x1,0x4,0x2,0x7,0x8,0x2,0x5,0xc,0xb,0x1,
        0xc,0xa,0x4,0xe,0xf,0x9,0xa,0x3,0x6,0xf,0x9,0x0,0x0,
        0x6,0xc,0xa,0xb,0xa,0x7,0xd,0xd,0x8,0xf,0x9,0x1,0x4,
        0x3,0x5,0xe,0xb,0x5,0xc,0x2,0x7,0x8,0x2,0x4,0xe
    },
    {
        0x2,0xe,0xc,0xb,0x4,0x2,0x1,0xc,0x7,0x4,0xa,0x7,0xb,
        0xd,0x6,0x1,0x8,0x5,0x5,0x0,0x3,0xf,0xf,0xa,0xd,0x3,
        0x0,0x9,0xe,0x8,0x9,0x6,0x4,0xb,0x2,0x8,0x1,0xc,0xb,
        0x7,0xa,0x1,0xd,0xe,0x7,0x2,0x8,0xd,0xf,0x6,0x9,0xf,
        0xc,0x0,0x5,0x9,0x6,0xa,0x3,0x4,0x0,0x5,0xe,0x3
    },
    {
        0xc,0xa,0x1,0xf,0xa,0x4,0xf,0x2,0x9,0x7,0x2,0xc,0x6,
        0x9,0x8,0x5,0x0,0x6,0xd,0x1,0x3,0xd,0x4,0xe,0xe,0x0,
        0x7,0xb,0x5,0x3,0xb,0x8,0x9,0x4,0xe,0x3,0xf,0x2,0x5,
        0xc,0x2,0x9,0x8,0x5,0xc,0xf,0x3,0xa,0x7,0xb,0x0,0xe,
        0x4,0x1,0xa,0x7,0x1,0x6,0xd,0x0,0xb,0x8,0x6,0xd
    },
    {
        0x4,0xd,0xb,0x0,0x2,0xb,0xe,0x7,0xf,0x4,0x0,0x9,0x8,
        0x1,0xd,0xa,0x3,0xe,0xc,0x3,0x9,0x5,0x7,0xc,0x5,0x2,
        0xa,0xf,0x6,0x8,0x1,0x6,0x1,0x6,0x4,0xb,0xb,0xd,0xd,
        0x8,0xc,0x1,0x3,0x4,0x7,0xa,0xe,0x7,0xa,0x9,0xf,0x5,
        0x6,0x0,0x8,0xf,0x0,0xe,0x5,0x2,0x9,0x3,0x2,0xc
    },
    {
        0xd,0x1,0x2,0xf,0x8,0xd,0x4,0x8,0x6,0xa,0xf,0x3,0xb,
        0x7,0x1,0x4,0xa,0xc,0x9,0x5,0x3,0x6,0xe,0xb,0x5,0x0,
        0x0,0xe,0xc,0x9,0x7,0x2,0x7,0x2,0xb,0x1,0x4,0xe,0x1,
        0x7,0x9,0x4,0xc,0xa,0xe,0x8,0x2,0xd,0x0,0xf,0x6,0xc,
        0xa,0x9,0xd,0x0,0xf,0x3,0x3,0x5,0x5,0x6,0x8,0xb
    }
};
```

等分密钥，密钥循环左移及密钥选取(代码如下)：

```
static ULONG8 keyleft[28]=
{
        57,49,41,33,25,17,9,1,58,50,42,34,26,18,
        10,2,59,51,43,35,27,19,11,3,60,52,44,36
};
static ULONG8 keyright[28]=
{
        63,55,47,39,31,23,15,7,62,54,46,38,30,22,
        14,6,61,53,45,37,29,21,13,5,28,20,12,4
};
static ULONG8 lefttable[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
static ULONG8 keychoose[48] ={
        14,17,11,24,1,5,3,28,15,6,21,10,
        23,19,12,4,26,8,16,7,27,20,13,2,
        41,52,31,37,47,55,30,40,51,45,33,48,
        44,49,39,56,34,53,46,42,50,36,29,32
};
```

(3) DES 密钥生成

DES 密钥是一个 64bit 的分组，但是其中 8bit 用于奇偶校验，所以密钥的有效位只有 56bit，由这 56bit 生成 16 轮子密钥。

首先将有效的 56bit 进行置换选择，将结果等分为 28bit 的两个部分，再根据所在的迭代轮数进行循环左移，左移后将两个部分合并为 56bit 的密钥，从中选取 48bit 作为此轮迭代的最终密钥，共生成 16 个 48bit 的密钥。每一个密钥，分为两个 24bit 的部分放在一个 ULONG32 的二维数组中保存。

每一轮密钥的生成，由 MakeKey 函数实现，具体程序如下所示。

```
INT32 MakeKey (ULONG32 * keyleft,ULONG32 * keyright ,ULONG32 number)
{
    ULONG32 tmpkey[2] ={0};
    ULONG32 * Ptmpkey= (ULONG32 * )tmpkey;
    ULONG32 * Poutkey= (ULONG32 * )&g_outkey[number];
    INT32 j;
    memset((ULONG8 * )tmpkey,0,sizeof(tmpkey));
     * Ptmpkey= * keyleft&leftandtab[lefttable[number]];
    Ptmpkey[1]= * keyright&leftandtab[lefttable[number]];
    if (lefttable[number]==1)
    {
        * Ptmpkey>>=27;
        Ptmpkey[1]>>=27;
    }
    else
    {
        * Ptmpkey>>=26;
        Ptmpkey[1]>>=26;
```

```
        }
        Ptmpkey[0] &= 0xfffffff0;
        Ptmpkey[1] &= 0xfffffff0;
        * keyleft<<=lefttable[number];
        * keyright<<=lefttable[number];
        * keyleft |=Ptmpkey[0];
        * keyright |=Ptmpkey[1];
        Ptmpkey[0]=0;
        Ptmpkey[1]=0;
        for (j=0; j<48; j++)
        {
            if (j<24)
            {

                if (* keyleft&pc_by_bit[keychoose[j]-1])
                {
                    Poutkey[0] |=pc_by_bit[j];
                }
            }
            else //j>=24
            {
                if (* keyright&pc_by_bit[(keychoose[j]-28)])
                {
                    Poutkey[1] |=pc_by_bit[j-24];
                }
            }
        }
        return SUCCESS;
    }
```

（4）DES 加密运算

DES 的加密运算也分为 16 圈迭代。

首先将明文分为 64bit 的数据块，不够 64bit 的用 0 补齐。在每一轮中，对每一个 64bit 的数据块，首先进行初始换位，并将数据块分为 32bit 的两部分，具体实现如下：

```
INT32   number=0 ,j=0;
ULONG32 * right=&left[1];
ULONG32 tmp=0;
ULONG32 tmpbuf[2]={ 0 };
for (j=0; j<64; j++)
{
    if (j<32)
    {
        if (pc_first[j]>32)
        {
            if (* right&pc_by_bit[pc_first[j]-1])
```

```
                {
                    tmpbuf[0] |=pc_by_bit[j];
                }
            }
            else
            {
                if (* left&pc_by_bit[pc_first[j]-1])
                {
                    tmpbuf[0] |=pc_by_bit[j];
                }
            }
        }
        else
        {
            if (pc_first[j]>32)
            {
                if (* right&pc_by_bit[pc_first[j]-1])
                {
                    tmpbuf[1] |=pc_by_bit[j];
                }
            }
            else
            {
                if (* left&pc_by_bit[pc_first[j]-1])
                {
                    tmpbuf[1] |=pc_by_bit[j];
                }
            }
        }
    }
    * left  =tmpbuf[0];
    * right=tmpbuf[1];
```

经过初始置换并且分组之后，将进行 DES 加密算法的核心部分。

首先，保持左部不变，将右部由 32bit 扩展成为 48bit，分别存在两个 ULONG32 类型的变量里，每个占 24bit，具体实现如下。

```
    for (j=0; j<48; j++)
    {
        if (j<24)
        {
            if (* right&pc_by_bit[des_E[j]-1])
            {
                exdes_P[0] |=pc_by_bit[j];
            }
        }
```

```
        else
        {
            if (* right&pc_by_bit[des_E[j]-1])
            {
                exdes_P[1] |=pc_by_bit[j-24];
            }
        }
    }
```

在将右部扩展成为48bit之后，与该轮的密钥进行异或操作，由于48bit分在一个ULONG32数组中的两个元素中，因此要进行两次异或操作，具体实现如下。

```
for (j=0; j<2; j++)
{
    exdes_P[j] ^=g_outkey[number][j];
}
```

在异或操作完成之后，对新的48bit进行压缩操作，即S盒。

将其每取6bit，进行一次操作，具体实现如下。

```
exdes_P[1]>>=8;
rexpbuf[7]= (ULONG8) (exdes_P[1]&0x0000003fL);
exdes_P[1]>>=6;
rexpbuf[6]= (ULONG8) (exdes_P[1]&0x0000003fL);
exdes_P[1]>>=6;
rexpbuf[5]= (ULONG8) (exdes_P[1]&0x0000003fL);
exdes_P[1]>>=6;
rexpbuf[4]= (ULONG8) (exdes_P[1]&0x0000003fL);
exdes_P[0]>>=8;
rexpbuf[3]= (ULONG8) (exdes_P[0]&0x0000003fL);
exdes_P[0]>>=6;
rexpbuf[2]= (ULONG8) (exdes_P[0]&0x0000003fL);
exdes_P[0]>>=6;
rexpbuf[1]= (ULONG8) (exdes_P[0]&0x0000003fL);
exdes_P[0]>>=6;
rexpbuf[0]= (ULONG8) (exdes_P[0]&0x0000003fL);
exdes_P[0]=0;
exdes_P[1]=0;
```

8个6bit的数据存在ULONG rexpbuf[8]中，然后进行数据压缩操作，每一个6bit经过运算之后输出4bit，因此最终输出的是压缩后的32bit数据(代码如下)。

```
* right=0;
for (j=0; j<7; j++)
{
    * right |=des_S[j][rexpbuf[j]];
    * right<<=4;
```

```
}
* right |=des_S[j][rexpbuf[j]];
```

对新的 32bit 数据，进行一次置换操作（代码如下）。

```
datatmp=0;
for (j=0; j<32; j++)
{
    if (* right&pc_by_bit[des_P[j]-1])
    {
        datatmp |=pc_by_bit[j];
    }
}
* right=datatmp;
```

再把左右部分进行异或作为右半部分，最原始的右边作为左半部分（代码如下）。

```
* right ^= * left;
* left=oldright;
```

最后进行逆初始置换，完成一轮完整的加密操作（代码如下）。

```
for (j=0; j<64; j++)
{
    if (j<32)
    {
        if (pc_last[j]>32)
        {
            if (* right&pc_by_bit[pc_last[j]-1])
            {
                tmpbuf[0] |=pc_by_bit[j];
            }
        }
        else
        {
            if (* left&pc_by_bit[pc_last[j]-1])
            {
                tmpbuf[0] |=pc_by_bit[j];
            }
        }
    }
    else
    {
        if (pc_last[j]>32)
        {
            if (* right&pc_by_bit[pc_last[j]-1])
            {
                tmpbuf[1] |=pc_by_bit[j];
```

```
                }
            }
            else
            {
                if (* left&pc_by_bit[pc_last[j]-1])
                {
                    tmpbuf[1] |=pc_by_bit[j];
                }
            }
        }
}
* left =tmpbuf[0];
* right=tmpbuf[1];
```

(5) 封装 DES 加密函数

将上述运算整合在一起,可以封装成一个加密函数,以便于调用,其中 pPlaintext 为明文部分, nPlaintextLength 为明文长度, pCipherBuffer 为准备存放密文的缓冲区, nCipherBufferLength 为密文长度,pKey 为密钥,nKeyLength 为密钥长度,代码如下:

```
    INT32 Encry(char * pPlaintext,int nPlaintextLength,char * pCipherBuffer,int
&nCipherBufferLength, char * pKey,int nKeyLength)
    {
        if(nKeyLength!=8)
        {
```

首先检查初始密钥长度,若正确,则创建 16 圈迭代的密钥(代码如下)。

```
    return 0;
}
MakeFirstKey((ULONG32 * )pKey);
```

由于加密、解密均要以 32bit 为单位进行操作,故需要计算相关参数,以确定加密的循环次数以及密文缓冲区是否够用,确定后将需要加密的明文格式化到新分配的缓冲区内(代码如下)。

```
int nLenthofLong=((nPlaintextLength+7)/8) * 2;
if(nCipherBufferLength<nLenthofLong * 4)
{//out put buffer is not enough
    nCipherBufferLength=nLenthofLong * 4;
    return 0;
}
memset(pCipherBuffer,0,nCipherBufferLength);
ULONG32 * pOutPutSpace=(ULONG32 * )pCipherBuffer;
ULONG32 * pSource;
if(nPlaintextLength!=sizeof(ULONG32) * nLenthofLong)
{
    pSource=new ULONG32[nLenthofLong];
```

```
        memset(pSource,0,sizeof(ULONG32) * nLenthofLong);
        memcpy(pSource,pPlaintext,nPlaintextLength);
    }
    else          {
        pSource= (ULONG32 * )pPlaintext;
    }
```

开始对明文进行加密,加密后将之前分配的缓冲区从内存中删除(代码如下)。

```
    ULONG32 gp_msg[2]= {0,0};
    for (int i= 0;i< (nLenthofLong/2);i+ + )
    {
        gp_msg[0]=pSource [2* i];
        gp_msg[1]=pSource [2* i+ 1];
        HandleData(gp_msg,DESENCRY);
        pOutPutSpace[2* i]=gp_msg[0];
        pOutPutSpace[2* i+ 1]=gp_msg[1];
    }
    if(pPlaintext!= (char * ) pSource)
    {
        delete []pSource;
    }

    return SUCCESS;
    }
```

最后需要说明,上述函数为一次完整的加密流程,解密流程与加密流程基本一致,仍是先进行初始置换,最后进行逆置换,中间 16 轮利用 16 个密钥的迭代加密,唯一不同的地方在于所生成的 16 个密钥的使用顺序,加密运算与解密运算的密钥使用顺序正好相反。

2. 基于 TCP 的聊天功能模块设计实现分析

TCP 通信流程如图 3-4 所示。

(1) 建立连接

对于客户端,首先输入服务器 IP 地址,建立并初始化连接套接字和 sockaddr_in 结构体,向服务器请求连接,进行实时聊天,关闭套接字(代码如下)。

```
    char strIpAddr[16];
    cin>> strIpAddr;
    int nConnectSocket, nLength;
    struct sockaddr_in sDestAddr;
    if ((nConnectSocket= socket(AF_INET, SOCK_STREAM, 0))< 0)
    {
        perror("Socket");
        exit(errno);
    }
    sDestAddr.sin_family= AF_INET;
```

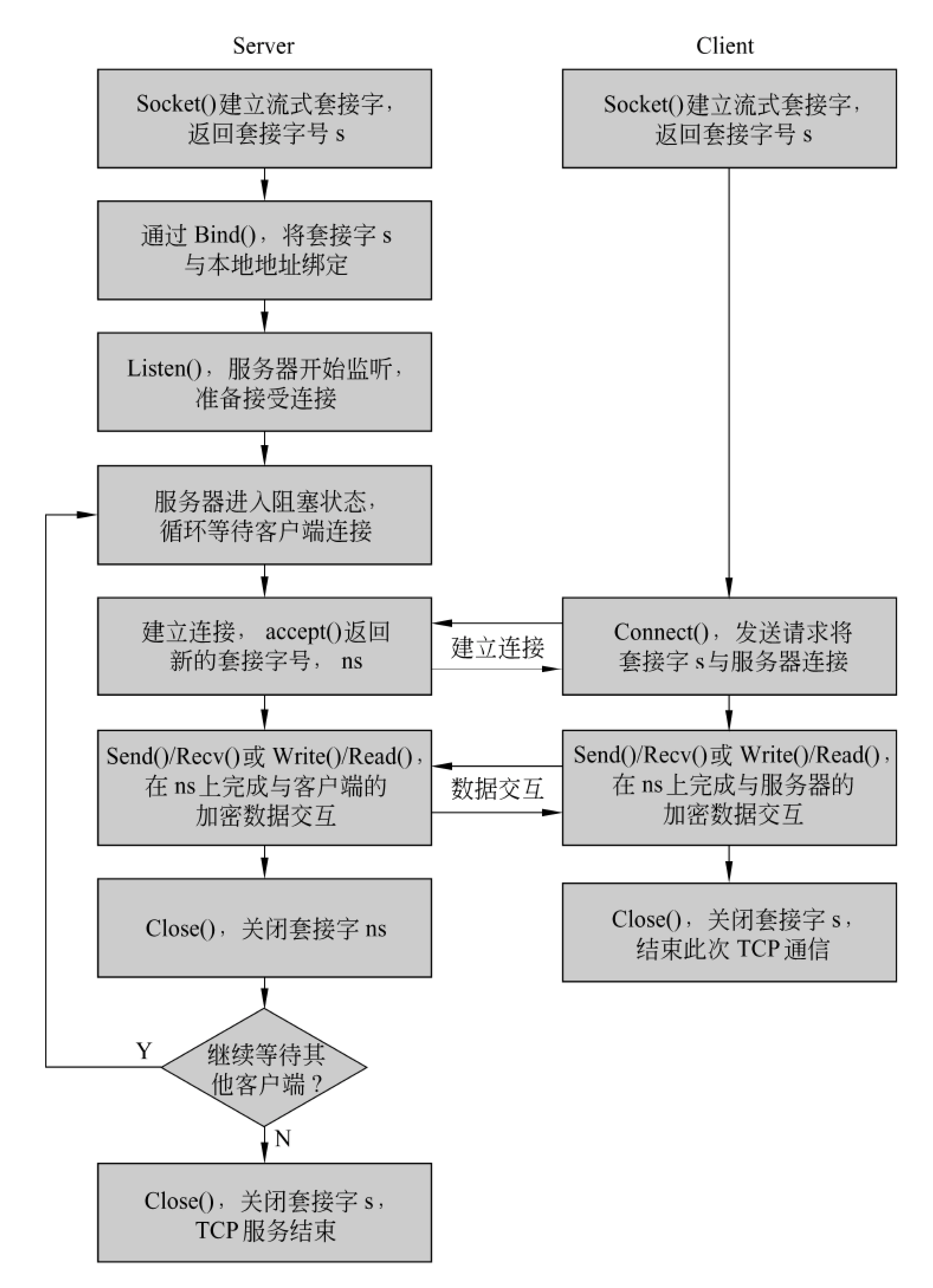


图 3-4 TCP 通信流程示意图

```
sDestAddr.sin_port=htons(SERVERPORT);
sDestAddr.sin_addr.s_addr=inet_addr(strIpAddr);
if (connect(nConnectSocket, (struct sockaddr * ) &sDestAddr, sizeof(sDestAddr))!=0)
{
    perror("Connect ");
    exit(errno);
}
else
{
    printf("Connect Success!  \nBegin to chat...\n");
    SecretChat(nConnectSocket,strIpAddr,"benbenmi");    }
close(nConnectSocket);
```

对于服务器端，建立并初始化本地 sockaddr_in 结构体，与本地套接字绑定并开始监听，建立远程 sockaddr_in 和套接字，在接受客户端连接请求后存储客户端的相关信息(代码如下)。

```
int nListenSocket, nAcceptSocket;
struct sockaddr_in sLocalAddr, sRemoteAddr;
…
if (bind(nListenSocket, (struct sockaddr * ) &sLocalAddr, sizeof(struct sockaddr))==-1)
{
    perror("bind");
    exit(1);
}
if (listen(nListenSocket, 5)==-1)
{
    perror("listen");
    exit(1);
}
nAcceptSocket=accept(nListenSocket,(struct sockaddr * ) &sRemoteAddr,&nLength);
close(nListenSocket);printf ("server: got connection from %s, port %d, socket %d\n",inet_ntoa
(sRemoteAddr.sin_addr),ntohs(sRemoteAddr.sin_port), nAcceptSocket);
SecretChat(nAcceptSocket,inet_ntoa(sRemoteAddr.sin_addr),"benbenmi");
close(nAcceptSocket);
}
```

(2) 多进程全双工聊天程序分析

Linux 是一种多用户、多进程的操作系统。每个进程都有一个唯一的进程标识符，操作系统通过对机器资源进行时间共享，并发地运行许多进程。

在 Linux 中，程序员可以使用 fork()函数创建新进程，它可以与父进程完全并发地运行。fork()函数不接受任何参数，并返回一个 int 值。当它被调用时，创建出的子进程除了拥有自己的进程标识符以外，其余特征，例如数据段、堆栈段、代码段等和其父进程完全相同。fork()函数向子进程返回 0，向父进程返回子进程的进程标识符，该标识符是一个非零的 int 值。

当 fork()函数被执行后，一个完全独立的子进程已经创建完毕并开始运行，在代码中可以利用该函数的返回值来区分父进程和子进程。另外，每个进程都有自己独立的堆栈段，所以两个进程的局部变量相互独立，在任意一个进程中都可以随便访问而不必考虑同步问题，但是如果进程使用了文件指针，则必须小心对待，因为两个进程的文件指针将会指向同一个底层文件，并行的读写操作可能造成冲突。

另外，如果调用 fork()函数的次数过于频繁造成系统中进程总数过多，系统可能由于耗尽所有可用的资源而导致创建新进程失败。

该聊天功能在函数 SecretChat()中实现，摘录代码如下。

```
void SecretChat(int nSock,char * pRemoteName, char * pKey)
{
    CDesOperate cDes;
```

```
if(strlen(pKey) != 8)
{
    printf("Key length error");
    return;
}

pid_t nPid;
nPid= fork();
if(nPid!= 0)
{
    while(1)
    {
        bzero(&strSocketBuffer, BUFFERSIZE);
        int nLength= 0;
        nLength= TotalRecv(nSock, strSocketBuffer,BUFFERSIZE,0);
        if(nLength != BUFFERSIZE)
        {
            break;
        }
        else
        {
            int nLen= BUFFERSIZE;
            cDes.Decry(strSocketBuffer,BUFFERSIZE,strDecryBuffer,nLen,pKey,8);
            strDecryBuffer[BUFFERSIZE- 1]= 0;
            if(strDecryBuffer[0] != 0&&strDecryBuffer[0] != '\n')
            {
                printf("Receive message form <%s>: %s\n", pRemoteName,
                strDecryBuffer);
                if(0==memcmp("quit",strDecryBuffer,4))
                {
                    printf("Quit!\n");
                    break;
                }
            }
        }
    }
}
else
{
    while(1)
    {
        bzero(&strStdinBuffer, BUFFERSIZE);
        while(strStdinBuffer[0]==0)
        {
            if (fgets(strStdinBuffer, BUFFERSIZE, stdin)==NULL)
```

```
                {
                    continue;
                }
            }
            int nLen=BUFFERSIZE;
            cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pKey,8);
            if(send(nSock, strEncryBuffer, BUFFERSIZE,0)!=BUFFERSIZE)
            {
                perror("send");
            }
            else
            {
                if(0==memcmp("quit",strStdinBuffer,4))
                {
                    printf("Quit!\n");
                    break;
                }
            }
        }
    }
```

该函数的参数有 3 个，其中 nSock 是 socket 句柄，要求其必须是一个已经建立连接的 socket；pRemoteName 指向一个字符串，代表远程主机的名字；pKey 指向另一个字符串，储存 DES 密钥。程序在连接建立完成后，直接调用该函数执行聊天功能，其中密钥为双方事先共享的字符串。

该函数在完成必要的错误检查后，调用 fork()函数创建了一个子进程，如果条件“if(nPid!=0)”满足，则代表当前进程为父进程，否则为子进程。父进程负责接收密文消息，解密并输出到屏幕；同时子进程负责从标准输入读取消息，加密并发送到指定的套接字，两个进程完全并行，实现实时聊天的功能。由于父、子进程的代码原理基本相同，故只对子进程代码进行分析。

聊天通信双方传递固定大小的数据块，子进程通过调用 fgets()函数从标准输入读取数据，如果无法读到数据，则始终循环等待，直到读取到用户输入为止。由于本代码做教学使用，为了简化编写，未考虑用户输入超过缓冲区长度的现象。

在获得用户输入后，调用类 CDesOperate 中封装的加密函数 Encry()进行加密，然后调用 send()函数将加密后的数据块发送出去。

此外，需要解释的是，在父进程中从 socket 接收数据使用函数 TotalRecv()，这是因为 TCP 在某些特殊情况下(例如链路质量变化)，recv()函数一次并不能返回对方发送的全部数据，需多次调用 recv()函数才能获得全部数据，使用该函数是为了确保将整个数据块可以被顺利接收。

该函数代码如下。

```
ssize_t TotalRecv(int s, void* buf, size_t len, int flags)
{
```

```
        size_t nCurSize=0;
        while(nCurSize < len)
        {
            ssize_t nRes=recv(s,((char*)buf)+nCurSize,len-nCurSize,flags);
            if(nRes<0||nRes+nCurSize>len)
            {
                return -1;
            }
            nCurSize+=nRes;
        }
        return nCurSize;
    }
```

3.4 扩展与提高

3.4.1 高级套接字函数

1. recv 函数和 send 函数

recv 函数和 send 函数提供了和 read 函数与 write 函数类似的功能，不同之处是前者提供了第 4 个参数来控制读写操作。两个函数的原型如下。

```
int recv(int sockfd,void* buf,int len,int flags)
int send(int sockfd,void* buf,int len,int flags)
```

该两函数的前 3 个参数和 read 函数、write 函数相同，第 4 个参数可以是 0 或者是以下的组合：

(1) MSG_DONTROUTE：不查找路由表；

(2) MSG_OOB：接受或者发送带外数据；

(3) MSG_PEEK：查看数据，并不从系统缓冲区移走数据；

(4) MSG_WAITALL：等待所有数据。

MSG_DONTROUTE：send 函数使用的标志。这个标志告诉 IP 协议，目的主机在本地网络上面，没有必要查找路由表。这个标志一般用在网络诊断和路由程序中。

MSG_OOB：表示可以接收和发送带外数据。理论上可以把带外数据看作是流套接字数据传输中独立的高优先级传输通道。带外数据是独立于普通数据传送给用户的，每次传输至少传送一个字节的带外数据。

MSG_PEEK：recv 函数的使用标志，表示只是从系统缓冲区中读取内容，而不清除系统缓冲区的内容。以便下次读的时候，仍然是相同的内容。一般在有多个进程读写数据时可以使用这个标志。

MSG_WAITALL：recv 函数的使用标志，表示等到所有的信息到达时才返回。使用这个标志的时候 recv 函数会一直阻塞，直到指定的条件满足，或者是发生了错误。

(1) 当读到了指定的字节时，函数正常返回，返回值等于 len；

（2）当读到了文件的结尾时，函数正常返回，返回值小于 len；

（3）当操作发生错误时，返回－1，且设置错误为相应的错误号（errno）。

如果 flags 为 0，则功能和 read 函数、write 函数完全相同。

2．shutdown 函数

shutdown 函数的原型如下。

```
int shutdown(int sockfd,int howto)
```

TCP 连接是双向的（即是可读写的），当使用 close 时，会把读写通道都关闭，有时候希望只关闭一个方向，这个时候可以使用 shutdown 函数。针对不同的 howto 参数，系统会采取不同的关闭方式。

（1）howto＝0 时，系统会关闭读通道，但是可以继续对套接字描述符进行写操作。

（2）howto＝1 时，和上面相反，系统会关闭写通道，此时只可以对套接字描述符进行读操作。

（3）howto＝2 时，关闭读写通道，功能和 close 相同。在多进程程序里面，有几个子进程共享一个套接字时，如果使用 shutdown 函数，此时所有的子进程都不能够操作，这个时候只能使用 close 来关闭子进程的套接字描述符。

3.4.2　新一代对称加密协议 AES

由于 DES 及其变形算法的安全强度已经难以继续满足新的安全需要，难以对抗 20 世纪末出现的差分和线性密码分析，而且其实现速度，代码大小以及跨平台性均难以满足新的需求，美国政府于 1997 年开始公开征集新的数据加密标准（Advanced Encryption Standard，AES）算法，以取代 DES。经过 3 轮筛选，最后选中比利时密码学家 Joan Daemen 和 Vincent Rijmen 提出的密码算法 Rijndael 作为 AES 正式取代 DES。

1．AES 算法描述

Rijndael 是具有可变分组长度和可变密钥长度的分组密码。其分组长度和密钥长度均可以独立地设定为 32bit 的任意倍数，最小值为 128bit，最大值为 256bit，其输入输出均可看做是一个一维的字符数组。假设输入明文为 $c_0, c_1, \cdots, c_i \cdots$，其中 $0<i<4*N$。将明文映射到一个字节矩阵上，称之为状态，在本例中 $N=4$，密钥同理可以映射到密钥状态中，如表 3-17 所示。

表 3-17　数据块长度为 128bit 的状态

c_0	c_4	c_8	c_{12}
c_1	c_5	c_9	c_{13}
c_2	c_6	c_{10}	c_{14}
c_3	c_7	c_{11}	c_{15}

Rijndael 加密算法由 3 个部分组成：一个初始轮密钥加法变换；$N-1$ 轮变换；最后一轮变换。下面给出实现该算法的伪代码：

```
Rijndael(State, CipherKey)
{
    //密钥扩展,即把输入的密钥扩展为加密用的密钥
    KeyExpansion(CipherKey, ExpandKey);
    //初始轮密钥加法变换
    AddRoundKey(State,ExpandKey);
    //N-1轮变换
    for (i=1;i<N;i++)
        Round(State, ExpandedKey[i])
        {
            //S-盒变换,非线性的置换操作
            ByteSub(State);
            //字节换位,将状态中的行按不同的偏移量进行循环移位
            ShiftRow(State);
            //作用在状态各列的置换操作
            MixColumn(State);
            //密钥加法
            AddRoundKey(State, ExpandedKey[i]);
        }
    //最后一轮变换
    FinalRound(State, ExpandedKey[N])
    {
        ByteSub(State);
        ShiftRow(State);
        AddRoundKey(State, ExpandedKey[i]);
    }
}
```

考虑到 AddRoundKey()函数可以通过在每一列上执行一个额外的 32bit 异或运算来实现,故也可以利用 4KB 的表通过查表操作实现,该实现方案对于每一轮的每一列仅仅需要 4 次查表和 4 次异或运算,实现这些操作的效率很高。同理,在解密算法中,也可以将轮变换的不同步骤合并为一组表的查询。

2. AES 算法与 DES 算法的比较

在一篇关于 AES 和 DES 两种算法比较的论文中,作者曾利用两种算法在相同环境下对同样长度的文件进行加密,并且对其加密效率和时间进行了比较,程序的环境为 Windows 2000,CPU 2.2GHz,内存 256MB,文件长度为 3.77MB(3960928B),AES 分组长度为 128bit,密钥长度为 128bit,DES 分组长度为 64bit,密钥长度为 56bit,具体实验结果如表 3-18 与表 3-19 所示。

表 3-18 AES 和 DES 加解密的时间比较

算法	加密(s)	解密(s)
DES	18	18
AES	6	10

表 3-19 AES 和 DES 加解密效率比较

算法	加密(Mbps)	解密(Mbps)
DES	1.676	1.676
AES	5.027	3.016

从上述实验数据可以看出，AES 算法的分组长度与密钥长度均大于 DES 算法，而加解密效率也高于 DES 算法，更为重要的是，用穷举法破解 AES 在有限的时间内是不可能的。上述所有因素促成了 AES 取代传统 DES 成为新一代的加密体系。

3.4.3　DES 安全性分析

自 DES 算法 1977 年首次公诸于世以来，学术界对其进行了深入的研究，围绕它的安全性展开了激烈的争论。

1. DES 的安全性缺陷

在技术上对 DES 的批评主要集中在以下 3 个方面：

(1) 作为分组密码，DES 的加密单位仅有 64bit 二进制，这对于数据传输来说太小，因为每个分组仅含有 8 个字符，而且其中某些位还要用于奇偶校验或其他通信开销。

(2) DES 的密钥太短，有效密钥只有 56bit，而且各次迭代中使用的密钥是递推产生的，这种相关性必然会降低密码体制的安全性，在现有技术下用穷举法寻找密钥已趋于可行。1999 年在电子前沿组织(Electronic Frontier Foundation，EFF)进行的一次测试中，只用了不到 3 天的时间就破解了一个 DES 加密系统。

(3) DES 不能对抗差分和线性密码分析。

2. 多重 DES 算法

针对 DES 算法上的缺陷，现在已发展出几十种改进的 DES 算法，经过比较，大多学者认为多重 DES 算法具有较高的可行性。为了增加密钥的长度，采用多重 DES 加密技术，将分组密码进行级联，在不同的密钥作用下，连续多次对一组明文进行加密。针对 DES 算法，最常用的是 3 重 DES 加密算法，它只用到了两个 56bit 的 DES 密钥。假设这两个密钥为 K_1 和 K_2，则该算法的步骤如下。

(1) 用密钥 K1 进行 DES 加密。

(2) 用步骤 1 的结果使用密钥 K2 进行 DES 解密。

(3) 用步骤 2 的结果使用密钥 K1 进行 DES 加密。

3 重 DES 算法可使加密密钥长度扩展到 128bit，其中有效位数 112bit。3 重 DES 的 112bit 密钥长度在可以预见的将来被认为是合适的、安全的，目前还没有找到针对此方案的攻击方法。因为要破译它可能需要尝试 256 个不同的 56bit 密钥直到找到正确的密钥。但是 3 重 DES 的时间是 DES 算法的 3 倍，时间开销较大。

3. 密钥管理

现代密码学的特征是算法可以公开。系统的安全管理者，要根据本系统实际所使用的密钥长度与其所保护的信息的敏感程度、重要程度以及系统实际所处安全环境的恶劣程度，在留有足够的安全系数的条件下来确定其密钥和证书更换周期的长短。同时，将已废弃的密钥和证书放入黑库归档，以备后用。密钥更换周期的正确安全策略是系统能够安全运行的保障，是系统的安全管理者最重要、最核心的日常工作任务。

第 4 章

基于RSA算法自动分配密钥的加密聊天程序

4.1　编程训练目的与要求

在讨论了对称加密算法 DES 原理与实现方法的基础上，本章将以典型的非对称的公钥密码 RSA 算法为例，以实现基于 TCP 协议的聊天程序加密为目标，系统地讨论公钥密码体系与 RSA 算法的基本工作原理与应用软件编程方法。

本章训练的主要目的是：

（1）理解 RSA 算法的基本工作原理。

（2）掌握基于 RSA 算法的网络加密通信系统设计方法与实现技术。

（3）掌握在 Linux 平台上实现 RSA 算法的编程方法。

（4）了解 Linux 操作系统异步 I/O 接口的基本工作原理。

本章编程训练的要求如下：

（1）要求在 Linux 平台上完成基于 RSA 算法的自动分配密钥加密聊天程序的编写。

（2）应用程序除要保持第 3 章“基于 DES 加密的 TCP 通信”中要求的全部功能以外，要在此基础上进行扩展，实现密钥自动生成，并基于 RSA 算法进行密钥共享。

（3）要求程序能够实现加密的全双工通信，并且加密过程对用户是透明的。

4.2　相关背景知识

1. 公钥密码体系的基本概念

传统对称密码体制要求通信双方使用相同的密钥，因此应用系统的安全性完全依赖于密钥的保密。针对对称密码体系的缺陷，Diffie 和 Hellman 提出了新的密码体系——公钥密码体系，也称为非对称密码体系。在公钥加密系统中，加密和解密使用两把不同的密钥。加密的算法和公钥可以公开，但是解密的私钥必须保密，只有解密方知道。公钥密码体系要求算法要能够保证：任何攻击者都无法从公钥中推算出私钥。

公钥密码体制中最著名算法是 RSA 算法，以及背包密码、McEliece、Diffie_Hellman、Rabin、零知识证明、椭圆曲线和 ElGamal 算法等。

2. 公钥密码体系的特点

公钥密码体系由如下几个部分组成：

(1) 明文：作为算法输入的消息或者数据。

(2) 加密算法：加密算法对明文进行各种代换和变换。

(3) 密文：作为算法的输出，看起来完全随机而杂乱的数据，依赖明文和密钥。对于给定的消息，不同的密钥将产生不同的密文，密文是随机的数据流，并且其意义是无法理解的。

(4) 公钥和私钥：公钥和私钥成对出现，一个用来加密，另一个用来解密。

(5) 解密算法：该算法用来接收密文，解密还原出明文。

(6) 公钥密码体系的基本结构如图 4-1 所示。

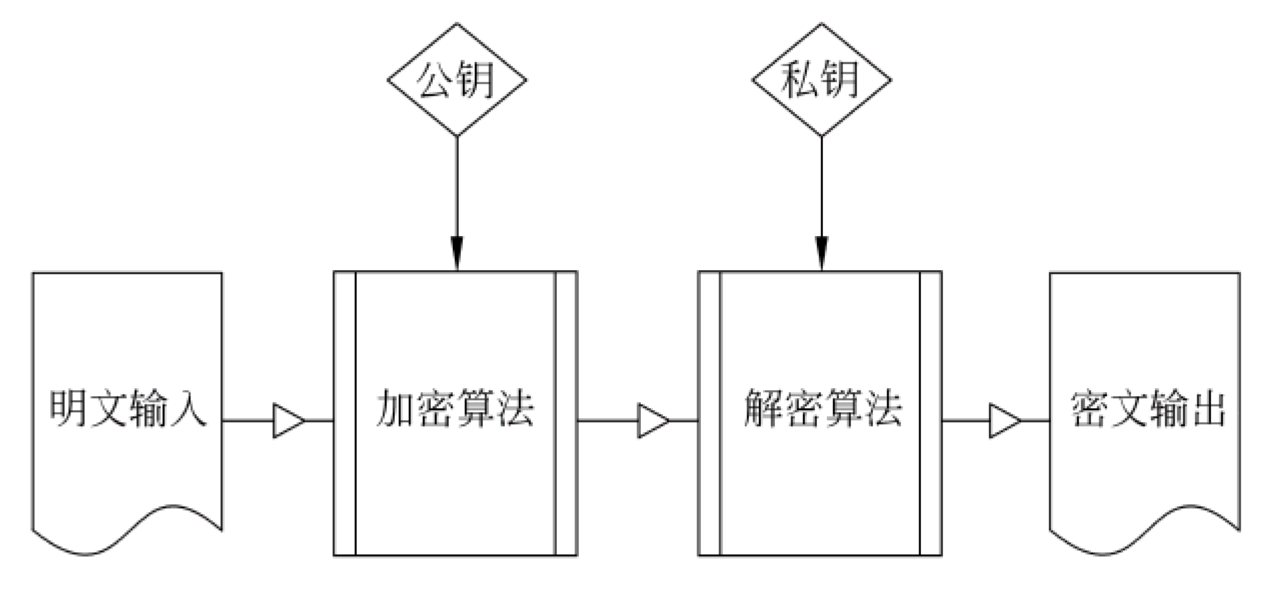


图 4-1　公钥密码体系原理示意图

3. RSA 加密算法的基本工作原理

RSA 加密算法是一种典型的公钥加密算法。RSA 算法的可靠性建立在分解大整数的困难性上。目前一般只有使用短密钥进行加密的 RSA 加密结果才可能被穷举法破译。只要其密钥的长度足够长，用 RSA 加密的信息的安全性就可以保证。

RSA 密码体系使用了乘方运算。明文以分组为单位进行加密，每个分组的二进制值均小于 n，即分组的大小必须小于或者等于 $\mathrm{Log}_2 n$。在实际应用中，分组的大小是 k 位，则 $2^k < n < 2^{k+1}$。

对于明文分组 M 和密文分组 C，其加密和解密的过程如下。

(1) $C = M^e \% n$

(2) $M = C^d \% n = (M^e)^d \% n = M^{d \times e} \% n = M$

其中 n、d、e 为 3 个整数，且 $d \times e \equiv 1 \% \phi(n)$。收发双方共享 n，接收一方已知 d，发送一方已知 e，则此算法的公钥为 $\{e, n\}$，私钥是 $\{d, n\}$。

理解 RSA 算法的基本工作原理需要数论的一些基础知识：

(1) 同余：两个整数 a、b，若它们除以整数 m 所得的余数相等，则称 a、b 对于模 m 同余，记作 $a \equiv b \% m$。

(2) Euler 函数：$\phi(n)$是指所有小于 n 的正整数里，和 n 互质的整数的个数。其中 n 是一个正整数。假设整数 n 可以按照质因数分解写成如下形式：$n = P_1^{a_1} \times P_2^{a_2} \times \cdots \times P_m^{a_m}$；其中，$P^1, P^2, \cdots, P^n$ 为质数。则 $\phi(n) = n \times \left(1 - \frac{1}{P^1}\right) \times \left(1 - \frac{1}{P^2}\right) \times \cdots \times \left(1 - \frac{1}{P^m}\right)$。

4. RSA 密码体系公钥与私钥生成方法

RSA 密码体系公钥与私钥生成方法如下：

(1) 任意选取两个质数 p 和 q，设 $n=p\times q$；

(2) 函数 $\phi(n)$ 为 Euler 函数，返回小于 n 且与 n 互质的正整数个数；

(3) 选择一个正整数 e，使其与 $\phi(n)$ 互质且小于 $\phi(n)$，公钥 $\{e,n\}$ 已经确定；

(4) 确定 d，使得 $d\times e\equiv 1\%\phi(n)$，即 $(d\times e-1)\%\phi(n)=0$。

至此，私钥 $\{d,n\}$ 也被确定。

4.3 实例编程练习

4.3.1 编程训练要求

本章训练要求读者在第 3 章“基于 DES 加密的 TCP 通信”的基础上进行二次开发，使原有程序可以实现自动生成 DES 密钥以及基于 RSA 算法的密钥分配。

(1) 要求在 Linux 平台上完成基于 RSA 算法的保密通信程序的设计与编写。

(2) 程序必须包含 DES 密钥自动生成、RSA 密钥分配以及 DES 加密通信等 3 个部分。

(3) 要求程序实现全双工通信，并且加密过程对用户是透明的。

(4) 有能力的同学可以使用 select 模型或者异步 I/O 模型对“基于 DES 加密的 TCP 通信”一章中的 socket 通信部分代码进行优化。

4.3.2 编程训练设计与分析

1. 程序总体流程

程序执行过程如下：

(1) 在客户端与服务器建立连接后，客户端首先生成一个随机的 DES 密钥，在第 3 章的程序里要求密钥长度为 64bit，所以使用长度为 8 的字符串充当密钥；同时，服务端生成一个随机的 RSA 公钥/私钥对，并将 RSA 公钥通过建立的 TCP 连接发送到客户端主机。

(2) 客户端主机在收到 RSA 公钥后，使用公钥加密自己生成的 DES 密钥，并将加密后的结果发送给服务器端。

(3) 服务器端使用自己保留的私钥解密客户端发过来的 DES 密钥，最后双方使用该密钥进行保密通信。

程序的流程图如图 4-2 所示。

2. 模乘运算和模幂运算

模乘运算即计算两个数的乘积然后取模，代码如下：

```
inline unsigned __int64 MulMod(unsigned __int64 a, unsigned __int64 b, unsigned __int64 n)
{
    return (a%n) * (b%n) %n;
}
```

程序为了提升运算速度，根据求模运算的性质 $\forall x,y,n|((x\%n)\times(y\%n))\%n=(x\times y)\%n$，优化了算法。

模幂运算即首先计算某数的若干次幂，然后对其结果进行取模运算，函数实现的代码

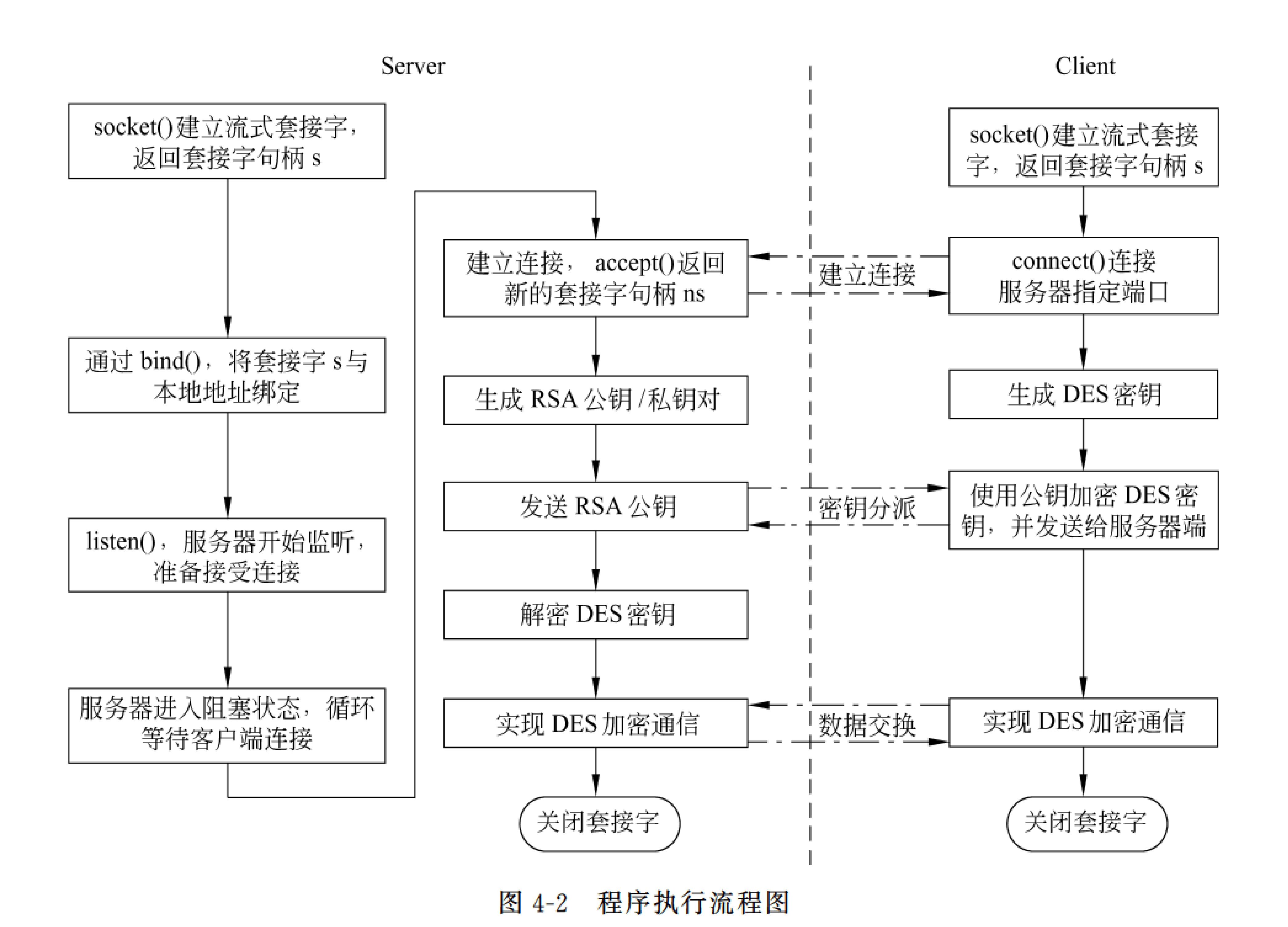


图 4-2　程序执行流程图

如下：

```
unsigned __int64 PowMod(unsigned __int64 base, unsigned __int64 pow, unsigned __int64 n)
{
    unsigned __int64    a=base, b=pow, c=1;
    while(b)
    {
        while(!(b & 1))
        {
            b>>=1;
            a=MulMod(a, a, n);
        }
        b--;
        c=MulMod(a, c, n);
    }
    return c;
}
```

由于 $\forall x,y,n \mid ((x\%n)\times(y\%n))\%n=(x\times y)\%n$，所以可以推出 $\forall x,z,n \mid ((x\%n)^z)\%n=(x^z)\%n$。此外 $\forall \alpha,\beta,\gamma \mid \gamma^{\alpha+\beta}=\gamma^\alpha\times\gamma^\beta, \gamma^{\alpha\times\beta}=\gamma^{\alpha\beta}$。

在代码中，笔者根据上述公式进行优化。变量 a、b 分别代表计算过程中未完成乘方运算的底数和指数，c 为运算中间变量，用于存储运算结果。程序使用外层 while 循环用来遍历代表未完成乘方操作的次数 b。在该循环内，首先考虑未完成的乘方次数是否为偶数，如果为偶数，设 $b=2\times k$，则根据上述讨论，$a^b\%n=a^{2\times k}\%n=a^{2k}\%n=(a^2\%n)^k\%n$。内层的

while 循环即完成此操作，每次循环，b 变为原来的$\frac{1}{2}$，而 a 变为 $a^2\%n$。

内层循环在未完成乘方次数 b 为奇数时跳出，程序继续处理此种情况。基于上述讨论，设 $b=2\times k+1$，则 $a^b\%n=a^{2\times k+1}\%n=((a^{2k}\%n)\times a\%n)\%n$。在程序中首先将 b 值减 1，然后将 $a\%n$ 的值暂存在变量 c 中，而最后一次模 n 的操作将在 $b=1$ 程序即将返回时执行。

故程序循环处理乘方操作，并将求模操作分布到乘方运算过程中，直到全部乘方操作运算完成，跳出外层循环并将结果返回。

3. 生成随机的大质数

本文使用基于 Fermat 定理"p 是一个质数，整数 a 与 p 互素，那么 $a^p\equiv1\%p$"的 Rabin-Miller 质数测试方法进行质数判断，该方法是一种基于概率的质数判别方法。利用此方法设计质数判别函数，其代码如下：

```
long RabinMillerKnl(unsigned __int64 &n)
{
    unsigned __int64    a, q, k, v;
    q=n-1;
    k=0;
    while(!(q&1))
    {
        ++k;
        q>>=1;
    }
```

程序首先计算出 q、k，使得 $n-1=q\times2\times k$，其中 q 是正奇数，k 是非负整数（代码如下）。

```
    a=2+cRadom.Random(n-3);
    v=PowMod(a, q, n);
    if(v==1)
    {
        return 1;
    }
```

随机取一个数 a，使 $2\leqslant a<n-1$，然后计算 $a^q\%n$，如果 $a^q\%n=1$，则通过测试，表示该数字可能是质数（代码如下）。

```
for(int j=0;j<k;j++)
{
    unsigned int z=1;
    for(int w=0;w<j;w++)
    {
        z*=2;
    }
    if(PowMod(a, z*q, n)==n-1)
```

```
        return 1;
    }
    return 0;
}
```

最后循环检验 $a^{zq}\%n(z=2^{j})$的值，如果该值等于 $n-1$，则证明此数字可能是一个质数，否则该数为合数。

但是，基于 Rabin-Miller 的检验方法为概率算法，一个奇合数有 1/4 的概率通过检验，故在实际应用中，需要通过多次重复检验，以增加验证正确的概率。

基于重复调用 Rabin-Miller 质数判别函数的代码如下：

```
long RabinMiller(unsigned __int64 &n, long loop=100)
{
    for(long i=0; i<loop; i++)
    {
        if(!RabinMillerKnl(n))
        {
            return 0;
        }
    }
    return 1;
}
```

在完成质数判断函数 Rabin-Miller 的设计后，进而实现质数生成函数 RandomPrime，其代码如下：

```
unsigned __int64 RandomPrime(char bits)
{
    unsigned __int64    base;
    do
    {
        base=(unsigned long)1 << (bits-1);          //保证最高位是 1
        base+=cRadom.Random(base);                  //再加上一个随机数
        base|=1;                                    //保证最低位是 1,即保证是奇数
    } while(!RabinMiller(base, 30));                //进行拉宾—米勒测试 30 次
return base;                                        //全部通过认为是质数
}
```

该函数首先生成一个确保最高位是一(确保足够大)的随机奇数，然后，检验该奇数是否是质数，如果该奇数不是质数，则重复该过程直到生成所需的质数为止。

4. 求最大公约数

求最大公约数的代码运用了欧几里得辗转相除法，其代码如下：

```
unsigned __int64 Gcd(unsigned __int64 &p, unsigned __int64 &q)
{
```

```
    unsigned __int64    a=p>q? p:q;
    unsigned __int64    b=p<q? p:q;
    unsigned __int64    t;
    if(p==q)
    {
        return p;                               //两数相等,最大公约数就是本身
    }
    else
    {
        while(b)                                //辗转相除法,gcd(a,b)=gcd(b,a-qb)
        {
            a=a%b;
            t=a;
            a=b;
            b=t;
        }
   return a;
    }
}
```

使用循环遍历法也可以计算两个数的公约数,但是运算效率较低。

5. 私钥生成

计算私钥主要就是计算 d 的值,根据第 3 节介绍的基础知识,d 必须满足 $(d\times e-1)\%\phi(n)=0$,所以,求 d(已知 e 和 $\phi(n)$)的过程等价于寻找二元方程 $e\times d-\phi(n)\times i=1$ 的最大整数解(i 为另一未知量)。

程序实现的代码如下:

```
unsigned __int64 Euclid(unsigned __int64 e, unsigned __int64 t_n)
{
    unsigned __int64 Max=0xffffffffffffffff-t_n;
    unsigned __int64 i=1;

    while(1)
    {
        if(((i*t_n)+1)%e==0)
        {
            return ((i*t_n)+1)/e;
        }
        i++;
        unsigned __int64 Tmp=(i+1)*t_n;
        if(Tmp>Max)
        {
            return 0;
        }
```

```
    }
    return 0;
}
```

程序在一个循环中不断从小到大尝试可能的 i 值，何时 $\phi(n)\times i+1$ 能被 e 整除，则返回对应的 d 值，程序返回；否则，一直不断尝试更大的值直到数据超过阈值，则返回 0，表示密钥生成失败。

6. 密钥分配

为了方便程序编写，在示例代码中把 DES 加密、解密和 RSA 加密、解密的代码分别封装在类 CDesOperate 和 CRSASection 中。

在 CRSASection 中导出 3 个函数，分别用来进行加密、解密以及导出密钥。

(1) 加密函数 Encry(代码如下)

```
static UINT64 Encry(unsigned short nSorce, PublicKey &cKey)
{
    return PowMod(nSorce, cKey.nE, cKey.nN);
}
```

此函数是加密函数，使用公钥，通过模幂运算实现计算 $C=M^e \bmod n$ 的加密过程，由于公钥本身并不始终保存在类的成员变量里，所以加密函数设计为 static，并通过参数传递公钥。

(2) 解密函数 Decry(代码如下)

```
unsigned short Decry(UINT64 nSorce)
{
    UINT64 nRes= PowMod(nSorce, m_cParament.d, m_cParament.n);
    unsigned short * pRes= (unsigned short * )&(nRes);
    if(pRes[1]!=0||pRes[3]!=0||pRes[2]!=0)
    {//error
        return 0;
    }
    else
    {
        return pRes[0];
    }
}
```

函数 Decry()用于进行解密计算，即实现 $M=C^d \bmod n$ 的计算。其中，密钥由保存在类成员变量中的结构实体 m_cParament 提供。

(3) 公钥获取函数 GetPublicKey(代码如下)

```
PublicKey GetPublicKey()
{
    PublicKey cTmp;
    cTmp.nE= this->m_cParament.e;
```

```
        cTmp.nN=this->m_cParament.n;
        return cTmp;
    }
```

本函数用来输出当前使用的公钥,其中 PublicKey 是一个结构体,用于保存公钥中的两个整数。

加密函数的输入和解密函数的输出为短整型变量,这是因为虽然理论上 RSA 可以加密,解密任意小于 n 的整数(n 为 64 位),但是在中间计算中仍可能生成大于 n 的临时变量。本程序为了简化编写并未使用专业大整数函数库,这就可能造成如果加密函数输入过大在进行乘法操作时溢出。故限制加密函数、解密函数的输入输出范围为短整型。

(4) 生成公钥私钥

在 socket 连接建立起来以后,首先服务端通过调用函数 RsaGetParam()初始化与 RSA 加密相关的各项参数,如公钥,私钥等。程序实现的代码如下:

```
RsaParam RsaGetParam(void)
{
    RsaParam            Rsa={ 0 };
    UINT64     t;
    Rsa.p=RandomPrime(16);                    //随机生成两个素数
    Rsa.q=RandomPrime(16);
    Rsa.n=Rsa.p* Rsa.q;
    Rsa.f=(Rsa.p-1) * (Rsa.q-1);
    do
    {
        Rsa.e=m_cRadom.Random(65536);
        Rsa.e|=1;
    } while(Gcd(Rsa.e, Rsa.f)!=1);
    Rsa.d=Euclid(Rsa.e, Rsa.f);
    Rsa.s=0;
    t=Rsa.n>>1;
    while(t)
    {
        Rsa.s++;
        t>>=1;
    }
    return Rsa;
}
```

这个函数会在类的构造函数中自动调用。

(5) DES 密钥分配

此后,程序会自动进行 DES 密钥的分配。

① 首先,服务端生成 RSA 密码的公钥与私钥,并将私钥通过 socket 传送到客户端。服务端程序实现的代码如下:

```
CRSASection cRsaSection;
```

```
cRsaPublicKey= cRsaSection.GetPublicKey();
if(send(nAcceptSocket,
        (char * )(&cRsaPublicKey), sizeof(cRsaPublicKey),0)!= sizeof(cRsaPublicKey))
{
    perror("send");
    exit(0);
}
else
{
    printf("successful send the RSA public key. \n");
}
UINT64 nEncryptDesKey[DESKEYLENGTH/2];
if(DESKEYLENGTH/2 * sizeof(UINT64) != TotalRecv(nAcceptSocket,
            (char * )nEncryptDesKey,DESKEYLENGTH/2 * sizeof(UINT64),0))
{
    perror("TotalRecv DES key error");
    exit(0);
}
else
{
    printf("successful get the DES key. \n");
    unsigned short * pDesKey= (unsigned short * )strDesKey;
    for(int i= 0;i< DESKEYLENGTH/2;i+ + )
    {
        pDesKey[i]= cRsaSection.Decry(nEncryptDesKey[i]);
    }
}
printf("Begin to chat...\n");
SecretChat(nAcceptSocket,inet_ntoa(sRemoteAddr.sin_addr),strDesKey);
close(nAcceptSocket);
```

② 客户端首先调用函数 GerenateDesKey()生成随机 DES 密钥,然后获得服务端提供的 RSA 公钥,并使用该公钥加密 DES 密钥,然后将加密后的 DES 密钥发回给服务端,从而实现 DES 密钥的可靠共享。客户端程序实现的代码如下:

```
printf("Connect Success!  \n");
GerenateDesKey(strDesKey);
printf("Create DES key success\n");
if(sizeof(cRsaPublicKey)= = TotalRecv(nConnectSocket, (char * )&cRsaPublicKey,
                    sizeof(cRsaPublicKey),0))
{
    printf("Successful get the RSA public Key\n");
}
else
{
    perror("Get RSA public key ");
```

```
        exit(0);
    }
    UINT64 nEncryptDesKey[DESKEYLENGTH/2];
    unsigned short * pDesKey= (unsigned short * )strDesKey;
    for(int i=0;i< DESKEYLENGTH/2;i++)
    {
        nEncryptDesKey[i]=CRSASection::Encry(pDesKey[i],cRsaPublicKey);
    }
    if(sizeof(UINT64) * DESKEYLENGTH/2!=send(nConnectSocket, (char * )nEncryptDesKey,
                            sizeof(UINT64) * DESKEYLENGTH/2,0))
    {
        perror("Send DES key Error");
        exit(0);
    }
    else
    {
        printf("Successful send the encrypted DES Key\n");
    }
    printf("Begin to chat...\n");
    SecretChat(nConnectSocket,strIpAddr,strDesKey);
```

(6) 加密全双工通信

最后调用函数 SecretChat()实现加密全双工通信。基于 RAS 算法的密钥分配增加了原有程序的安全性。攻击者只能通过监听截获 RSA 公钥和使用 RSA 公钥加密后的 DES 密钥,却无法获得对应的 RSA 私钥,故无法解密 DES 密钥,进而可以保证 DES 加密通信的安全性。

此外,由于 RSA 算法执行的运算量较大,所以只使用 RSA 算法用于密钥共享,而不是直接使用其加密通信内容,以降低系统资源的消耗。

4.4 扩展与提高

4.4.1 RSA 安全性

RSA 算法是第一个能同时用于加密和数字签名的算法,也易于理解和操作。同时它也是应用范围最广的公钥密码体系,从提出到现在已近二十年,经历了各种攻击的考验,逐渐为人们所接受。目前,公众普遍认为 RSA 是最优秀的公钥方案之一。但是 RSA 的安全性依赖于大数的因子分解,但并没有从理论上证明破译 RSA 的难度与大数分解难度等价。即无法从理论上证明不可能存在一种不需要进行大数分解即可以破解 RSA 的算法,即 RSA 的重大缺陷是无法从理论上把握它的保密性能如何,而且密码学界多数人士倾向于因子分解不是 NPC 问题。目前也有很多科学家在相关领域进行研究。

目前,对 RSA 算法的攻击有以下 3 种常用算法:

(1) 穷举攻击:试图穷举所有可能的私钥;

(2) 数学攻击:方法多种多样,但其本质就是试图分解 n,求得 p 和 q,进而推算出密钥;

(3) 计时攻击：这类算法依赖于观测解密算法的运行时间，攻击者从算法运行时间中获得额外信息进行攻击。

对抗穷举攻击，RSA 也是使用大的密钥空间，所以进行选择时 d 和 e 越大越好，但是密钥产生过程和加密解密过程都需要经过复杂的运算，所以这两个数选择得越大，加密解密所需的时间就越长，需要程序员在两者之间选择最佳的平衡点。

数学攻击主要指因子分解攻击，即分解 n 为两个质因子，从而计算出 $\phi(n)=(p-1)\times(q-1)$，根据公式 $d\times e\equiv 1\%\phi(n)$，进一步根据 e 确定 d 的取值，从而猜测密钥，进行破解。

目前，虽然分解具有大质数因子的 n 仍然是一个难题，但是已经逐渐被科学界攻破，现在破解固定长度公钥所需要的时间正在逐年递减，所以，要想确保 RSA 算法的安全性，就必须保证 n 足够大，此外 RSA 的发明者建议 p 和 q 应满足下列条件：

(1) p 和 q 的长度应仅相差几位。

(2) $(p-1)$和$(q-1)$都应该有一个大的质因子。

(3) $(p-1)$和$(q-1)$的最大公约数应该比较小。

此外，已经证明若 $e<n$ 且 $d>n/4$，则 d 容易被确定。

至于计时攻击，是一种通过记录计算机解密消息所用时间来确定私钥的一种攻击方式，类似于观察他人转动保险柜拨号盘的时间长短来猜测密码。计时攻击并非仅针对 RSA 算法，而是可以攻击全部的公钥密码体系，所以其危害比较严重。例如，在攻击 RSA 算法时，因为在进行加密时所进行的模指数运算是逐位进行的，而位为 1 所花的运算比位为 0 的运算要多得多，故其通过观察得到多组信息与其加密时间，就可以尝试反推出私钥的内容。

程序编写者可以使用一些简单的办法来防御此类攻击：

(1) 保证所有的幂运算在返回结果之前所用的时间相同。

(2) 在求幂算法中加入随机延时。

(3) 通过执行幂运算前将密文乘上一个随机数进行隐藏。

总之，就目前的技术水平来说，分解 n 依然是针对 RSA 算法最主要的攻击方法。现在，现有技术水平已经能分解 140 位(十进制)的大素数。因此，模数 n 必须尽量选择大数，才能保证 RSA 算法的安全性。在实际应用中，1997 年后开发的系统，已经使用了 1024 位密钥，而安全要求较高的证书认证机构采用 2048 位或以上密钥。

4.4.2　其他公钥密码体系

椭圆曲线密码学(Elliptic Curve Cryptography，ECC)是基于椭圆曲线数学的一种公钥密码方法。椭圆曲线在密码学中的使用是在 1985 年由 Neal Koblitz 和 Victor Miller 分别独立提出的。

椭圆曲线密码体制来源于对椭圆曲线的研究，所谓椭圆曲线指的是由韦尔斯特拉斯(Weierstrass)方程所确定的平面曲线。其并非真的椭圆曲线，只是因为其方程形式类似求解椭圆形周长的公式故得其名，椭圆曲线密码体制中用到的椭圆曲线都是定义在有限域上的，即最终方程形式如下：$y^2 \bmod p=(x^3+ax+b) \bmod p$。

首先定义椭圆曲线加法运算规则：若椭圆曲线上的 3 个点在同一条直线上，则其和为零。

设两点 P 和 Q，则在方程 $kP=Q$ 中，已知 k 和点 P 求点 Q 比较容易，反之已知点 Q 和

点 P 求 k 却很困难，这个问题称为椭圆曲线上点群的离散对数问题。椭圆曲线密码体制正是利用这个困难问题设计而来。

椭圆曲线密码体制是目前已知的公钥体制中，对每比特所提供加密强度最高的一种体制。解椭圆曲线上的离散对数问题的最好算法是 Pollard rho 方法，其时间复杂度是完全指数阶的。当密钥大小为 150 时，破解需要 3.8×10^{10} MIPS。MIPS 表示每秒钟处理的百万级的机器语言指令数，是衡量 CPU 速度的一个指标。当密钥增大到 234 时，破解时间就可达到 1.6×10^{28} MIPS 年(MIPS 年是指每秒运行百万条指令的处理器运行一年的计算量)。而大家熟知的 RSA 算法所利用的是大整数分解的困难问题，目前对于一般情况下的因数分解的最好算法的时间复杂度是子指数阶的，当密钥长度为 512 时，只需要 3×10^{4} MIPS 年，即便密钥长度增长到 2048，破解时间也不过增加到 3×10^{20} MIPS 年。即当 RSA 的密钥使用 2048 位时，ECC 的密钥使用 234 位所获得的安全强度比 RSA 密钥的安全强度要高出许多。但它们之间的密钥长度却相差 9 倍，当 ECC 的密钥更大时它们之间的差距将更大。可见 ECC 密钥短的优点是非常明显的。

国家标准与技术局和 ANSI X9 已经设定了最小密钥长度的要求，RSA 和 DSA 是 1024 位，ECC 是 160 位，相应的对称分组密码的密钥长度是 80 位。NIST 已经公布了一系列推荐的椭圆曲线用来保护 5 个不同的对称密钥大小(80、112、128、192、256)。一般而言，二进制域上的 ECC 需要的非对称密钥的大小是相应的对称密钥大小的 2 倍。

在 2005 年 2 月 16 日，NSA 宣布决定采用椭圆曲线密码的战略作为美国政府标准的一部分，用来保护敏感但不保密的信息。

4.4.3 使用 Select 机制进行并行通信

1. Linux select I/O 操作方式简介

为了提升程序效率，Linux 提供了 select 函数接口，用以同时管理若干个套接字或者句柄上的 I/O 操作，通过该 API，程序可以同时监控多个 socket 或者句柄上的 I/O 操作，进而可以免去开启多个进程的系统开销。函数定义如下。

```
int select(int n fd_set * readfds,fd_set * writefds,fd_set * exceptfds,
struct timeval * timeout)
```

其中，参数 n 代表最大文件句柄，必须赋值为监控的全部句柄中的最大值加一，参数 readfds、writefds 和 exceptfds 对应 3 个句柄集合，用来通知系统分别监控发生在对应集合中所包括句柄上的读、写或错误输出事件。

下面的一组宏用于操作上述句柄。

(1) FD_CLR(inr fd,fd_set * set)：用来清除句柄集合 set 中相关 fd 的项；

(2) FD_ISSET(int fd,fd_set * set)：用来测试句柄集合 set 中相关 fd 的项是否为真；

(3) FD_SET(int fd,fd_set * set)：用来设置句柄集合 set 中相关 fd 的项；

(4) FD_ZERO(fd_set * set)：用来清除句柄集合 set 的全部项。

参数 timeout 为结构 timeval，用来设置函数 select()的等待时间，其结构定义如下。

```
struct timeval
{
time_t tv_sec;
time_t tv_usec;
};
```

如果参数 timeout 设为 NULL，则表示 select()函数一直等待不会超时。

函数执行成功则返回文件句柄状态已改变的个数，如果返回 0 则代表在句柄状态改变前已超时，当有错误发生时则返回－1，并将错误原因存于 errno 中。

(1) EBADF：文件句柄为无效的或该文件已关闭；

(2) EINTR：此调用被中断；

(3) EINVAL：参数 n 为无效；

(4) ENOMEM：核心内存不足。

2. 使用 select 优化函数 SecretChat

使用 select()函数重写后的 SecretChat()函数代码如下。

```
void SecretChat(int nSock,char * pRemoteName, char * pKey)
{
    CDesOperate cDes;
    fd_set cHandleSet;
    struct timeval tv;
    int nRet;
```

初始化相关变量，并在一个循环中监控套接字和标准输入(代码如下)。

```
while(1)
{
    FD_ZERO(&cHandleSet);
    FD_SET(nSock, &cHandleSet);
    FD_SET(0, &cHandleSet);
    tv.tv_sec=1;
    tv.tv_usec=0;
    nRet=select(nSock>0? nSock+1:1, &cHandleSet, NULL, NULL, &tv);
```

这里可见，程序只监控套接字和标准输入上的读操作。并分别进行处理；此外，每循环一次，程序就必须重新设置句柄集合中的内容(代码如下)。

```
if(nRet<0)
{
    printf("Select ERROR!\n");
    break;
}
if(0==nRet)
{
```

```
        continue;
    }
```

在排除超时和出错的可能后，程序通过下面的两个 if 语句分别判断套接字和标准输入上是否发生了 I/O 操作，如有发生，则调用 recv()函数进行读取，并处理获得的数据。

```
    if(FD_ISSET(nSock,&cHandleSet))
    {
        bzero(&strSocketBuffer, BUFFERSIZE);
        int nLength=0;
        nLength=TotalRecv(nSock, strSocketBuffer,BUFFERSIZE,0);
        if(nLength !=BUFFERSIZE)
        {
            break;
        }
        else
        {
            int nLen=BUFFERSIZE;

            cDes.Decry(strSocketBuffer,BUFFERSIZE,strDecryBuffer,nLen,pKey,8);
            strDecryBuffer[BUFFERSIZE-1]=0;
            if(strDecryBuffer[0]!=0&&strDecryBuffer[0]!='\n')
            {
                printf("Receive message form <%s>: %s\n", pRemoteName,strDecryBuffer);
                if(0==memcmp("quit",strDecryBuffer,4))
                {
                    printf("Quit!\n");
                    break;
                }
            }
        }
    }

    if(FD_ISSET(0,&cHandleSet))
    {
        bzero(&strStdinBuffer, BUFFERSIZE);
        while(strStdinBuffer[0]==0)
        {
            if (fgets(strStdinBuffer, BUFFERSIZE, stdin)==NULL)
            {
                continue;
            }
        }
        int nLen=BUFFERSIZE;
        cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pKey,8);
        if(send(nSock, strEncryBuffer, BUFFERSIZE,0)!=BUFFERSIZE)
```

```
        {
                perror("send");
            }
            else
            {
                if(0==memcmp("quit",strStdinBuffer,4))
                {
                    printf("Quit!\n");
                    break;
                }
            }
        }
    }
}
```

4.4.4　使用异步 I/O 进行通信优化

1. 同步 I/O 操作和异步 I/O 操作的比较

Linux 还提供了一种异步 I/O 机制对程序效率进行优化。一般同步 I/O 调用会在系统 I/O 操作完成后返回，在该 I/O 未能完成时调用函数会将调用进程挂起等待；而异步 I/O 调用会在系统调用后直接返回，在系统内核真正完成调用后，通过消息或者回调函数通告调用进程本次 I/O 的执行结果。图 4-3 和 4-4 给出了 Linux 系统同步 I/O 与异步 I/O 操作的区别。

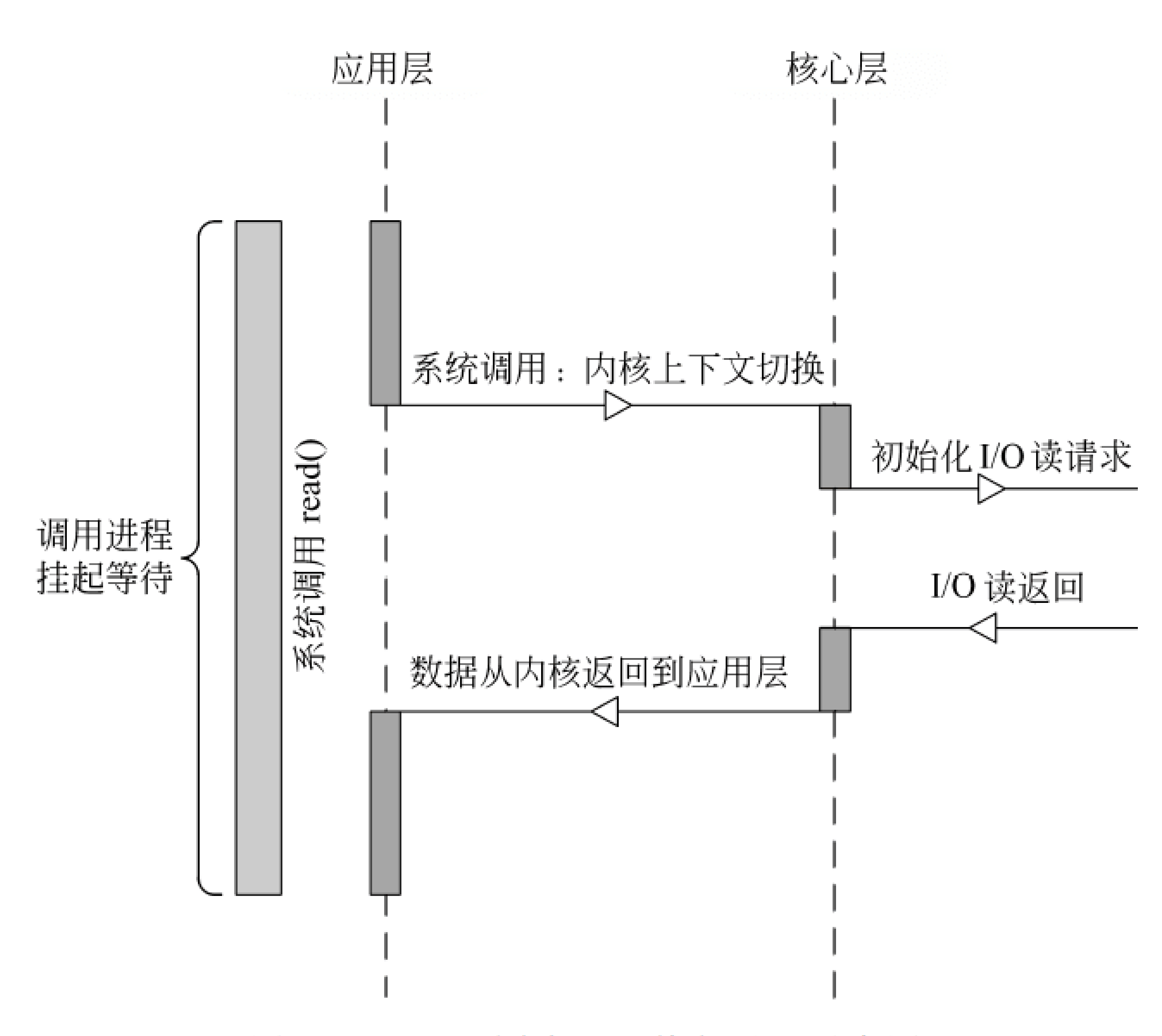


图 4-3　Linux 同步 I/O 执行过程示意图

使用异步 I/O 需要包含头文件＜aio. h＞，并在编译时指定编译选项 g＋＋ chat. cpp-lrt。

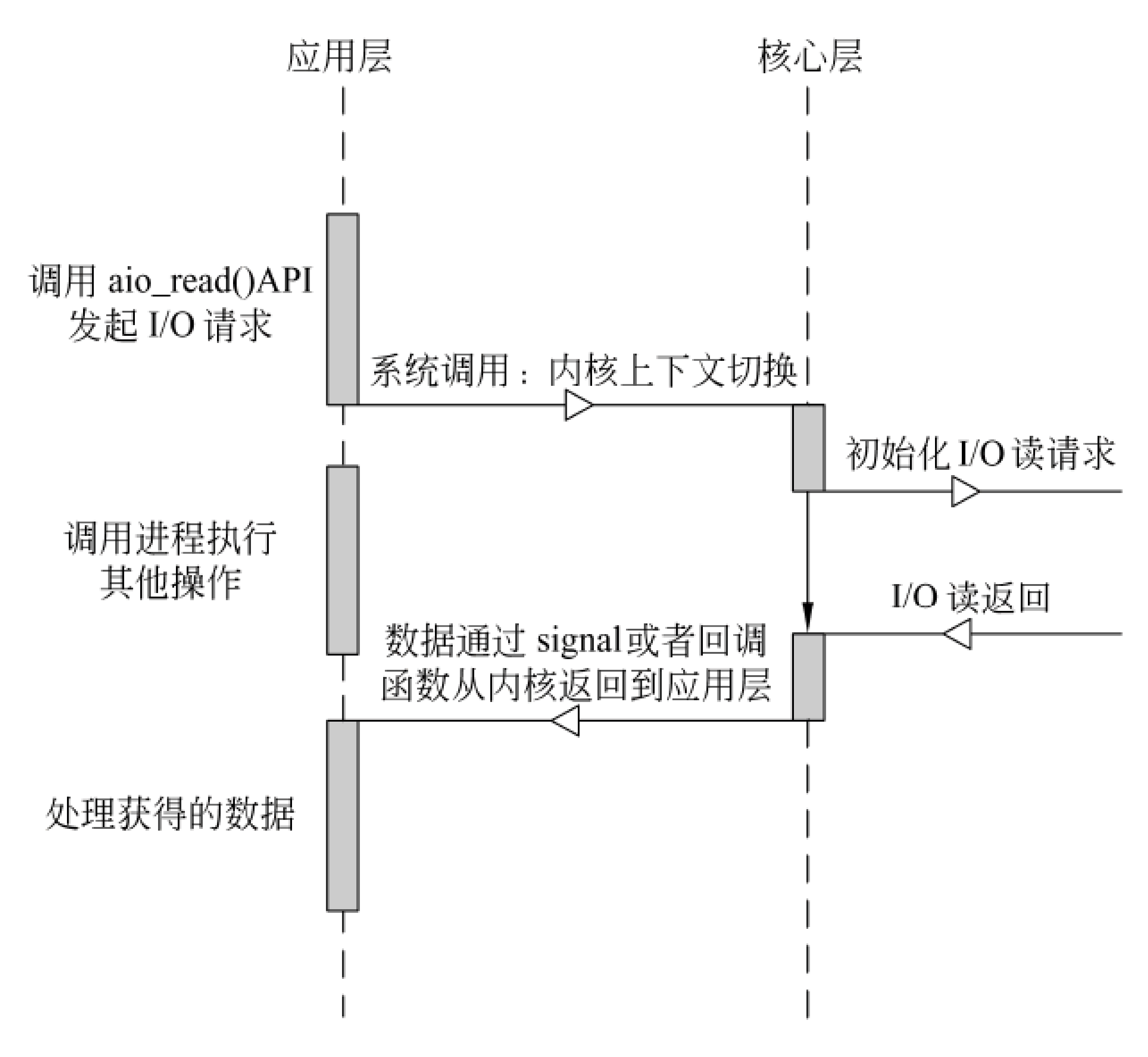


图 4-4　Linux 异步 I/O 执行过程示意图

异步 I/O 调用主要需要以下 API 函数：

(1) aio_read()：请求异步读操作；

(2) aio_error()：检查异步请求的状态；

(3) aio_return()：获得完成的异步请求的返回状态；

(4) aio_write()：请求异步写操作；

(5) aio_suspend()：挂起调用进程，直到一个或多个异步请求已经完成(或失败)；

(6) aio_cancel()：取消异步 I/O 请求；

(7) lio_listI/O()：发起一系列 I/O 操作。

其中，由于在异步非阻塞 I/O 中，程序可以同时发起多个传输操作。所以程序要求每个传输操作都有唯一的上下文，以便在收到内核 I/O 完成通知时区分该 I/O 通知是来自哪个 I/O 请求。这个工作由 aiocb 结构完成。该结构包含了有关传输的所有信息，包括为数据准备的用户缓冲区、通知的方式、回调函数的地址和参数等。在调用 I/O 请求 API 时，程序为该结构体赋值，并将该结构体的地址以参数形式传入内核。

2. 使用异步 I/O 优化函数 SecretChat

下述示例程序就是通过异步 I/O 进一步优化函数 SecretChat()的执行效率。

```
void SecretChat(int nSock,char * pRemoteName, char * pKey)
{
    pid_t nPid;
    nPid= fork();
    sem_t bStop;
    sem_init(&bStop,0,0);
    if(nPid!= 0)
    {
```

```
            SockinSingle* pSockin=new SockinSingle(pKey,nSock,bStop);
            aio_read(pSockin->m_pReq);
            sem_wait(&bStop);
        }
        else
        {
            StdinSingle* pStdin=new StdinSingle(pKey,nSock,bStop);
            aio_read(pStdin->m_pReq);
            sem_wait(&bStop);
        }
        sem_destroy(&bStop);
    }
```

程序创建了两个进程，分别在标准输入和 socket 上发起一个异步读操作，然后就通过 sem_wait()函数等待程序结束。所有的数据操作都在对应 I/O 结束后的回调函数中进行。

由于监控 socket 和监控标准输入的两个进程工作流程基本一致。所以在此只分析监控标准输入的进程。监控 socket 的相关代码可参考随书光盘的内容。

监控标准输入的进程由类 StdinSingle 完成，代码如下。

```
class StdinSingle
{
public:
    StdinSingle(char* pKey,int nSock,sem_t &bStop)
        :m_bStop(bStop)
    {
        memcpy(m_strKey,pKey,8);
        m_strKey[8]=0;
        this->m_nSock=nSock;
        bzero(&strStdinBuffer, BUFFERSIZE);
        this->m_pReq=new aiocb;
        bzero((char*)m_pReq, sizeof(struct aiocb));
        m_pReq->aio_fildes=0;
        m_pReq->aio_buf=strStdinBuffer;
        m_pReq->aio_nbytes=BUFFERSIZE;
        m_pReq->aio_offset=0;
        m_pReq->aio_sigevent.sigev_notify=SIGEV_THREAD;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._function=
        StdinReadCompletionHandler;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._attribute=NULL;
        m_pReq->aio_sigevent.sigev_value.sival_ptr=this;
    };
    ~StdinSingle()
    {
        delete m_pReq;
        bzero(&strStdinBuffer, BUFFERSIZE);
```

```
    };
    char m_strKey[9];
    int m_nSock;
    aiocb * m_pReq;
    sem_t &m_bStop;
    static void StdinReadCompletionHandler(sigval_t sigval)
    {
        StdinSingle * pThis= (StdinSingle * )sigval.sival_ptr;
        if (aio_error(pThis->m_pReq)==0)
        {
            int nSize=aio_return(pThis->m_pReq);
            CDesOperate cDes;
            int nLen=BUFFERSIZE;
            cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pThis->
            m_strKey,8);
            SockoutSingle * pSockoutSingle=new SockoutSingle(
                                                  pThis->m_nSock,
                                                  pThis->m_strKey,
                                                  pThis->m_bStop);
            aio_write(pSockoutSingle->m_pReq);
            if(0==memcmp("quit",strStdinBuffer,4))
            {
                printf("Quit!\n");
                sem_post(&pThis->m_bStop);
                exit(0);
            }
        }
        delete pThis;
        return;
    };
};
```

这个类主要在构造函数中初始化相关的 aiocb 结构，并指定其回调函数指针为 StdinReadCompletionHandler。

在该函数中，程序处理从标准输入读到的数据，并将其加密，然后通过 SockoutSingle 类调用 aio_write(pSockoutSingle→m_pReq)将加密后的数据 È 发送出去。

SockoutSingle 类的代码如下。

```
class SockoutSingle
{
public:
    char m_strKey[9];
    int m_nSock;
    aiocb * m_pReq;
    sem_t &m_bStop;
```

```
    SockoutSingle(int nSock,char * pKey,sem_t &bStop)
        :m_bStop(bStop)
    {
        memcpy(m_strKey,pKey,8);
        m_strKey[8]=0;
        this->m_nSock=nSock;
        this->m_pReq=new aiocb;
        bzero((char * )m_pReq, sizeof(struct aiocb));
        m_pReq->aio_fildes=nSock;
        m_pReq->aio_buf=strEncryBuffer;
        m_pReq->aio_nbytes=BUFFERSIZE;
        m_pReq->aio_offset=0;
        m_pReq->aio_sigevent.sigev_notify=SIGEV_THREAD;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._function=
        SockoutReadCompletionHandler;
        m_pReq->aio_sigevent._sigev_un._sigev_thread._attribute=NULL;
        m_pReq->aio_sigevent.sigev_value.sival_ptr=this;
    };
    ~SockoutSingle()
    {
        delete m_pReq;
    };
    static void SockoutReadCompletionHandler(sigval_t sigval)
    {
        SockoutSingle * pThis=(SockoutSingle * )sigval.sival_ptr;
        if (aio_error(pThis->m_pReq)==0)
        {
            int nSize=aio_return(pThis->m_pReq);
            if(nSize!=BUFFERSIZE)
            {
                perror("Error Send!\n");
            }
            else
            {
                StdinSingle * pStdin=new StdinSingle(pThis->m_strKey,
                                                    pThis->m_nSock,
                                                    pThis->m_bStop);
                aio_read(pStdin->m_pReq);
            }
        }
        delete pThis;
        return;
    }
};
```

这个类的工作流程和 StdinSingle 基本相同，其工作主要在回调函数 SockoutReadCompletionHandler 中完成。

当加密数据 Ê 发送完成后，函数 SockoutReadCompletionHandler()被调用，该函数在标准输入上发起读操作，从而驱动类 StdinSingle 继续工作。标准输入读入数据后会再次调用类 StdinSingle 中指定的回调函数 StdinReadCompletionHandler()。如此循环，直到用户输入“quit”命令，类 StdinSingle 通过 sem_post(&pThis→m_bStop)设置信号量，通知主进程结束，并自行退出。

异步模型的执行效率优势如何体现呢？每次 Linux 系统调用就会在内核和用户态之间进行一次上下文切换，需要消耗系统资源，使用异步 I/O 模型，两个主进程在发起第一次读操作后就通过 sem_wait()函数等待，直到结束直接退出，整个过程中都不需要进行上下文切换，而真正进行数据操作的回调函数也都是在 I/O 状态变化的时刻由内核自动调用的。可见，这个模型可以消除无谓进程上下文切换所需的资源，进而大大提升系统的执行效率。

此外，当系统同时处理若干个 socket 上的并发数据时，异步 I/O 可以使单个进程具有监督多个 socket 上数据的能力，从而大量节约所需进程的数目，大幅降低所需要的系统资源，提升程序效率。

第 5 章 基于MD5算法的文件完整性校验程序

5.1 本章训练目的与要求

MD5 是目前最流行的信息摘要算法，已广泛应用于数字签名、文件完整性检测等领域。熟悉 MD5 算法对于开发安全的网络应用程序具有重要的意义。

本章训练的主要目的是：

(1) 理解 MD5 算法的基本原理。

(2) 掌握利用 MD5 算法生成数据摘要的计算方法。

(3) 掌握将 MD5 算法应用于文件完整性校验软件的基本设计与编程方法。

(4) 掌握在 Linux 操作系统中检测文件完整性的基本方法。

本章编程训练的要求如下：

(1) 正确地实现 MD5 算法的计算过程。

(2) 对于任意长度的字符串能够生成 128 位 MD5 摘要。

(3) 对于任意长度的文件能够生成 128 位 MD5 摘要。

(4) 通过检查 MD5 摘要来检验原文件的完整性。

5.2 相关背景知识

5.2.1 MD5 算法的主要特点

MD5(Message-Digest Algorithm 5)是一种信息摘要算法。信息摘要算法的研究由来已久，早在 1990 年 Ronald L. Rivest 就提出了与 MD5 属于同一系列的散列函数 MD4。经过大量的密码学分析与不断的攻击检测，Rivest 于 1991 年提出了 MD4 的改进算法 MD5。目前 MD5 已经得到了广泛的应用，成为许多机构和组织的散列函数标准。

作为散列函数，MD5 有以下两个重要特性：

(1) 任意两组数据经过 MD5 运算后，生成相同摘要的概率极小。

(2) 即使在算法与程序已知的情况下，也不能从 MD5 摘要中反推出原始数据。

MD5 的典型应用就是为一段信息(message)生成信息摘要(message-digest)，以防止传输的数据被篡改。Linux 系统自带了计算和校验 MD5 摘要的工具程序——md5sum。用户可以在命令行终端直接运行该程序。md5sum 程序可以创建一个与原始文件名称相同，后缀为.md5 的文件，并将原始文件的 MD5 摘要保存在该文件中(代码如下所示)。对于文件 nankai.txt，利用 md5sum 程序计算出它的摘要(3c771aea5b7e191408ab6b0372ecbf0c)并保

存在文件 nankai.md5 中。

```
[root@ localhost MD5]# cd md5sum/
[root@ localhost md5sum]# ls
nankai.txt
[root@ localhost md5sum]# md5sum nankai.txt> nankai.md5
[root@ localhost md5sum]# ls
nankai.md5   nankai.txt
[root@ localhost md5sum]# vim nankai.md5
  1 3c771aea5b7e191408ab6b0372edbf0c   nankai.txt
...
```

同时，md5sum 程序可以对文件的完整性进行检测。它通过重新计算原始文件的 MD5 摘要，将计算结果与同名的.md5 文件中的摘要进行对比，若相同，则说明文件完整；否则，说明原文件受到了破坏。如下所示，程序共包含了两次检验。第 1 次通过比较 nankai.md5 中的摘要来验证文件 nankai.txt 是完整的。第 2 次修改了 nankai.txt 文件的内容(在标题后面增加了 written by sky)，导致校验失败。

```
[root@ localhost md5sum]# md5sum - c nankai.md5
nankai.txt: OK
[root@ localhost md5sum]# vim nankai.txt
  1 About Nankai University      (written by sky)
  2
  3 A key multidisciplinary and research- oriented university directly under the jurisdiction
of the Ministry of Education, Nankai University, located in Tianjin on the border of the sea of Bohai, is
also the alma mater of our beloved late Premier Zhou Enlai.

[root@ localhost md5sum]# md5sum - c nankai.md5
nankai.txt: FAILED
md5sum: WARNING: 1 of 1 computed checksum did NOT match
```

5.2.2 MD5 算法分析

MD5 算法分为 3 个步骤：消息的填充与分割、消息块的循环运算和摘要的生成。

1. 消息的填充与分割

MD5 算法以 512bit 为单位对输入消息进行分组。每个分组是一个 512bit 的数据块。同时每个数据块又由 16 个 32bit 的子分组构成。摘要的计算过程就是以 512bit 数据块为单位进行的。

在 MD5 算法中，首先需要对输入消息进行填充，使其比特长度对 512 求余的结果等于 448。即消息的比特长度将被扩展至 $N\times512+448$，即 $N\times64+56$ 个字节，其中 N 为正整数。

具体的填充方法是：在消息的最后填充一位 1 和若干位 0，直到满足上述条件时才停止用 0 对消息进行填充。然后在填充部分的后面用一个 64 位二进制数表示填充前消息的长

度。经过这两步的处理，现在消息的比特长度 $=N\times512+448+64=(N+1)\times512$。因为长度恰好是 512 的整数倍，所以在下一步中可以方便地对消息进行分组运算。

2. 消息块的循环运算

MD5 算法包含 4 个初始向量、5 种基本运算和 4 个基本函数。

(1) 初始向量

MD5 算法中有 4 个 32bit 的初始向量。它们分别是：$\boldsymbol{A}$=0x01234567，$\boldsymbol{B}$=0x89abcdef，$\boldsymbol{C}$=0xfedcba98，$\boldsymbol{D}$=0x76543210。

(2) 基本运算

5 种基本运算是指：

① $\tilde{\boldsymbol{X}}$：$\boldsymbol{X}$ 的逐 bit 逻辑“非”运算；

② $\boldsymbol{X}\wedge\boldsymbol{Y}$：$\boldsymbol{X}$、$\boldsymbol{Y}$ 的逐 bit 逻辑“与”运算；

③ $\boldsymbol{X}\vee\boldsymbol{Y}$：$\boldsymbol{X}$、$\boldsymbol{Y}$ 的逐 bit 逻辑“或”运算；

④ $\boldsymbol{X}\oplus\boldsymbol{Y}$：$\boldsymbol{X}$、$\boldsymbol{Y}$ 的逐 bit 逻辑“异或”运算；

⑤ $\boldsymbol{X}<<<s$：$\boldsymbol{X}$ 循环左移 s 位。

(3) 基本函数

MD5 算法中的 4 个非线性基本函数分别用于 4 轮计算。它们分别是：

① $F(x,y,z)=(x\wedge y)\vee(\tilde{x}\wedge z)$

② $G(x,y,z)=(x\wedge z)\vee(y\wedge\tilde{z})$

③ $H(x,y,z)=x\oplus y\oplus z$

④ $I(x,y,z)=y\oplus(x\vee\tilde{z})$

在设置好 4 个初始向量以后，就进入 MD5 的循环过程。循环过程就是对每一个消息分组的计算过程。每一次循环都会对一个 512bit 消息块进行计算，因此循环的次数就是消息中 512bit 分组的数目，即 $(N+1)$。

在一次循环开始时，首先要将初始向量 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$、$\boldsymbol{D}$ 中的值保存到向量 $\boldsymbol{A}_0$、$\boldsymbol{B}_0$、$\boldsymbol{C}_0$、$\boldsymbol{D}_0$ 中，然后再继续后面的操作，如下式所示。

$$\boldsymbol{A}_0=\boldsymbol{A}\quad \boldsymbol{B}_0=\boldsymbol{B}\quad \boldsymbol{C}_0=\boldsymbol{C}\quad \boldsymbol{D}_0=\boldsymbol{D}$$

MD5 循环体中包含了 4 轮计算（MD4 只有 3 轮），每一轮计算进行 16 次操作。每一次操作可概括如下：

① 从向量 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$、$\boldsymbol{D}$ 中任意选取 3 个向量作一次非线性函数运算。

② 将所得结果与剩下的第 4 个变量、一个 32bit 子分组 $X[k]$和一个常数 $T[i]$相加。

③ 将所得结果循环左移 s 位，并从向量 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$、$\boldsymbol{D}$ 中选取一个与之相加；最后用该结果取代剩余三者之一的值。

在第 1 轮计算中，如果用表达式 FF[abcd, k, s, i]表示如下的计算过程，则第 1 轮的 16 次操作可以表示为：

$$a=b+((a+I(b,c,d)+X[k]+T[i])<<<s)$$

```
FF[ABCD, 0, 7, 1]    FF [DABC, 1, 12, 2]    FF [CDAB, 2, 17, 3]     FF [BCDA, 3, 22, 4]
FF [ABCD, 4, 7, 5]   FF [DABC, 5, 12, 6]    FF [CDAB, 6, 17, 7]     FF [BCDA, 7, 22, 8]
FF [ABCD, 8, 7, 9]   FF [DABC, 9, 12, 10]   FF [CDAB, 10, 17, 11]   FF [BCDA, 11, 22, 12]
```

FF [ABCD, 12, 7, 13] FF [DABC, 13, 12, 14] FF [CDAB, 14, 17, 15] FF [BCDA, 15, 22, 16]

在第 2 轮计算中，如果用表达式 GG[abcd, k, s, i]表示如下的计算过程，则第 2 轮的 16 次操作可以表示为：

$$a=b+((a+G(b,c,d)+X[k]+T[i])<<<s)$$

GG [ABCD, 1, 5, 17]	GG [DABC, 6, 9, 18]	GG [CDAB, 11, 14, 19]	GG [BCDA, 0, 20, 20]
GG [ABCD, 5, 5, 21]	GG [DABC, 10, 9, 22]	GG [CDAB, 15, 14, 23]	GG [BCDA, 4, 20, 24]
GG [ABCD, 9, 5, 25]	GG [DABC, 14, 9, 26]	GG [CDAB, 3, 14, 27]	GG [BCDA, 8, 20, 28]
GG [ABCD, 13, 5, 29]	GG [DABC, 2, 9, 30]	GG [CDAB, 7, 14, 31]	GG [BCDA, 12, 20, 32]

在第 3 轮计算中，如果用表达式 HH[abcd, k, s, i]表示如下的计算过程，则第 3 轮的 16 次操作可以表示为：

$$a=b+((a+H(b,c,d)+X[k]+T[i])<<<s)$$

HH [ABCD, 5, 4, 33]	HH [DABC, 8, 11, 34]	HH [CDAB, 11, 16, 35]	HH [BCDA, 14, 23, 36]
HH [ABCD, 1, 4, 37]	HH [DABC, 4, 11, 38]	HH [CDAB, 7, 16, 39	HH [BCDA, 10, 23, 40]
HH [ABCD, 13, 4, 41]	HH [DABC, 0, 11, 42]	HH [CDAB, 3, 16, 43]	HH [BCDA, 6, 23, 44]
HH [ABCD, 9, 4, 45]	HH [DABC, 12, 11, 46]	HH [CDAB, 15, 16, 47]	HH [BCDA, 2, 23, 48]

在第 4 轮计算中，如果用表达式 II[abcd, k, s, i]表示如下的计算过程，则第 4 轮的 16 次操作可以表示为：

$$a=b+((a+I(b,c,d)+X[k]+T[i])<<<s)$$

II [ABCD, 0, 6, 49]	II [DABC, 7, 10, 50]	II [CDAB, 14, 15, 51]	II [BCDA, 5, 21, 52]
II [ABCD, 12, 6, 53]	II [DABC, 3, 10, 54]	II [CDAB, 10, 15, 55]	II [BCDA, 1, 21, 56]
II [ABCD, 8, 6, 57]	II [DABC, 15, 10, 58]	II [CDAB, 6, 15, 59]	II [BCDA, 13, 21, 60]
II [ABCD, 4, 6, 61]	II [DABC, 11, 10, 62]	II [CDAB, 2, 15, 63]	II [BCDA, 9, 21, 64]

在上面式子中，$X[k]$表示一个 512bit 消息块中的第 k 个 32bit 的子分组。即一个 512bit 数据块由 16 个 32bit 的子分组构成 。常数 $T[i]$表示 4294967296×abs(sin(i))的整数部分。4294967296 是 2 的 32 次方，abs(sin(i))是对 i 的正弦取绝对值，其中 i 以弧度为单位。

如下式所示，在 4 轮计算结束后，将向量 **A**、**B**、**C**、**D** 中的计算结果分别与向量 $\boldsymbol{A}_0$、$\boldsymbol{B}_0$、$\boldsymbol{C}_0$、$\boldsymbol{D}_0$ 相加，最后将结果重新赋给向量 **A**、**B**、**C**、**D**。

$$\boldsymbol{A}=\boldsymbol{A}_0+\boldsymbol{A}\quad \boldsymbol{B}=\boldsymbol{B}_0+\boldsymbol{B}\quad \boldsymbol{C}=\boldsymbol{C}_0+\boldsymbol{C}\quad \boldsymbol{D}=\boldsymbol{D}_0+\boldsymbol{D}$$

至此，一个 512bit 消息块的运算过程已经完成。MD5 算法将通过不断地循环，计算所有的消息块，直到处理完最后一块消息分组为止。

3. 摘要的生成

如下式所示，将 4 个 32bit 向量 **A**、**B**、**C**、**D** 按照从低字节到高字节的顺序拼接成一个 128bit 的摘要。

$$MD5(M)=\boldsymbol{A}\parallel\boldsymbol{B}\parallel\boldsymbol{C}\parallel\boldsymbol{D}$$

5.3　实例编程练习

5.3.1　编程练习要求

本章要求读者完成以下训练内容：

（1）要求在 Linux 平台上编写应用程序，正确地实现 MD5 算法。

（2）要求程序不仅能够为任意长度的字符串生成 MD5 摘要，而且可以为任意大小的文件生成 MD5 摘要。

（3）程序还可以利用 MD5 摘要验证文件的完整性。验证文件的完整性有两种方式：一种是在手动输入 MD5 摘要的条件下，计算出当前被测文件的 MD5 摘要，再将两者进行比对；另一种是先利用 Linux 系统工具 md5sum 为被测文件生成一个后缀为. md5 的同名文件，然后让程序计算出被测文件的 MD5 摘要，将其与. md5 文件中的 MD5 摘要进行比较，最后得出检测结果。

具体要求有以下几点：

1. 程序的输入格式

程序为命令行程序，可执行文件名为 MD5. exe，命令行格式如下：

```
./MD5 [选项] [被测文件路径] [.md5 文件路径]
```

其中[选项]是程序为用户提供的各种功能。在本程序中[选项]包括{-h，-t，-c，-v，-f}5 个基本功能。[被测文件路径]为应用程序指明被测文件所在文件系统中的路径。[. md5 文件路径]为应用程序指明由被测文件生成的. md5 文件所在文件系统中的路径。其中第一个参数为必选项，后两个参数可以根据功能进行选择。

2. 程序的执行过程

（1）打印帮助信息

在控制台命令行中输入. /MD5 -h，打印程序的帮助信息。如下所示，帮助信息详细地说明了程序的选项和执行参数。用户可以通过查询帮助信息充分了解程序的功能。

```
[root@ localhost MD5]# ./MD5 -h
MD5: usage:  [-h]   --help information
                 [-t]   --test MD5 application
                 [-c]   [file path of the file computed]
                        --compute MD5 of the given file
                 [-v]   [file path of the file validated]
                        --validate the integrality of a given file by manual input MD5 value
                 [-f]   [file path of the file validated]   [file path of the .md5 file]
                        --validate the integrality of a given file by read MD5 value from .md5
                        --file
```

（2）打印测试信息

在控制台命令行中输入. /MD5 -t，打印程序的测试信息。如下所示，测试信息是指本

程序对特定字符串输入所生成的 MD5 摘要。所谓特定的字符串是指在 MD5 算法官方文档(RFC1321)中给出的字符串。同时,该文档也给出了这些特定字符串的 MD5 摘要。因此只需要将本程序的计算结果与文档中的正确摘要进行比较,就可以验证程序的正确性。

```
[root@ localhost MD5]# ./MD5 -t
MD5("")=d41d8cd98f00b204e9800998ecf8427e
MD5("a")=0cc175b9c0f1b6a831c399e269772661
MD5("abc")=900150983cd24fb0d6963f7d28e17f72
MD5("message digest")=f96b697d7cb7938d525a2f31aaf161d0
MD5("abcdefghijklmnopqrstuvwxyz")=c3fcd3d76192e4007dfb496cca67e13b
MD5("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789")
=d174ab98d277d9f5a5611c2c9f419d9f
MD5("12345678901234567890123456789012345678901234567890123456789012345678901234567890")
=57edf4a22be3c955ac49da2e2107b67a
```

(3) 为指定文件生成 MD5 摘要

在控制台命令行中输入./MD5 -c [被测文件路径],计算出被测文件的 MD5 摘要并打印出来。如下所示,被测文件 nankai.txt 与可执行文件 MD5 处于同一个目录中。

```
[root@ localhost MD5]# ./MD5 -c nankai.txt
The MD5 value of file("nankai.txt") is
3c771aea5b7e191408ab6b0372ecbf0c
```

(4) 验证文件完整性方法 1

在控制台命令行中输入./MD5 -c [被测文件路径],程序会先让用户输入被测文件的 MD5 摘要,然后再重新计算被测文件的 MD5 摘要,最后将两个摘要逐位比较。若一致,则说明文件是完整的,否则,则说明文件遭到了破坏。整个执行过程如下所示。

```
[root@ localhost MD5]# ./MD5 -v nankai.txt
Please input the MD5 value of file("nankai.txt")...
abcdefghijklmnopqrstuvwxyz123456
The old MD5 value of file("nankai.txt") you have input is
abcdefghijklmnopqrstuvwxyz123456
The new MD5 value of file("nankai.txt") that has computed is
3c771aea5b7e191408ab6b0372ecbf0c
Match Error!The file has been modified!
[root@ localhost MD5]# ./MD5 -v nankai.txt
Please input the MD5 value of file("nankai.txt")...
3c771aea5b7e191408ab6b0372ecbf0c
The old MD5 value of file("nankai.txt") you have input is
3c771aea5b7e191408ab6b0372ecbf0c
The new MD5 value of file("nankai.txt") that has computed is
3c771aea5b7e191408ab6b0372ecbf0c
OK!The file is integrated
```

(5) 验证文件完整性方法 2

在控制台命令行输入 ./MD5 -f [被测文件路径] [.md5 文件路径],程序会自动读取

.md5 文件中的摘要，然后再重新计算出被测文件的 MD5 摘要，最后将两者逐位比较。若一致，则说明文件是完整的，否则，则说明文件遭到了破坏。整个执行过程如下所示。

```
[root@ localhost MD5]# ./MD5 - f nankai.txt nankai.md5
The old MD5 value of file("nankai.txt") in nankai.md5 is
3c771aea5b7e191408ab6b0372edbf0c
The new MD5 value of file("nankai.txt") that has computed is
3c771aea5b7e191408ab6b0372edbf0c
OK!The file is integrated
```

总之，本章编程训练有两个重点：一是掌握 MD5 算法，能够正确地编程实现该算法；二是理解在 Linux 平台上运用 MD5 算法检测文件完整性的过程，能够编程模拟这一过程。

5.3.2　编程训练设计与分析

程序可以分为两个主要部分：MD5 算法实现与文件完整性检验。其中前者为程序的核心部分，通过 MD5 类来实现。MD5 类可以为任意长度的消息生成 128bit 的 MD5 摘要。文件完整性检验包括读取被测文件，调用 MD5 类运算函数生成摘要和通过比较摘要判断文件完整性等工作。

1. MD5 类的设计与实现

作为程序的核心部分，MD5 类负责摘要计算的全部过程，并对外提供各种接口。它的定义如下：

```
class MD5
{
public:
    MD5();
    MD5(const string &str);
    MD5(ifstream &in);
    //对给定长度的输入流进行 MD5 运算
    void Update(const void* input,size_t length);
    //对给定长度的字符串进行 MD5 运算
    void Update(const string &str);
    void Update(ifstream &in);                      //对文件中的内容进行 MD5 运算
    const BYTE* GetDigest();                        //将 MD5 摘要以字节流的形式输出
    string Tostring();                              //将 MD5 摘要以字符串形式输出
    void Reset();                                   //重置初始变量

private:
    //对给定长度的字节流进行 MD5 运算
    void Update(const BYTE* input,size_t length);
    void Stop();                                    //用于终止摘要计算过程,输出摘要
    void Transform(const BYTE block[64]);           //对消息分组进行 MD5 运算
    //将双字流转换为字节流
```

```
    void Encode(const DWORD* input, BYTE* output, size_t length);
    //将字节流转换为双字流
    void Decode(const BYTE* input, DWORD* output, size_t length);
    //将字节流按照十六进制字符串形式输出
    string BytesToHexString(const BYTE* input, size_t length);

private:
    DWORD state[4];                    //用于表示 4个初始向量
    DWORD count[2];                    //用于计数,count[0]表示低位,count[1]表示高位
    BYTE  buffer_block[64];            //用于保存计算过程中按块划分后剩下的 bit 流
    BYTE  digest[16];                  //用于保存 128 比特长度的摘要
    bool  is_finished;                 //用于标志摘要计算过程是否结束

    static const BYTE padding[64];     //用于保存消息后面填充的数据块
    static const char hex[16];         //用于保存十六进制的字符
};
```

数组 state 表示 4 个初始向量。数组 count 是一个计数器,记录已经运算的 bit 数。buffer_block 是一个 64 字节的缓存块,保存消息被划分后不足 64 字节的数据。digest 用于保存生成的 MD5 摘要。is_finished 标志 MD5 运算是否结束。

Update 函数将不同类型的输入划分为若干个 64 字节的分组,然后调用 Transform 函数进行 MD5 运算。Transform 函数对一个 512bit 的消息分组进行 MD5 运算。Encode 函数和 Decode 函数实现消息分组在字节类型与双字类型之间的相互转换。Reset 函数用于重置初始向量。在对新消息进行 MD5 运算之前,需要把初始向量设为默认值。GetDigest 函数用于获得 MD5 摘要。BytesToHexString 函数将 MD5 摘要转换为十六进制字符串形式。Tostring 函数将 MD5 摘要以十六进制字符串的形式输出。其中 Update 函数和 Transform 函数是 MD5 类的核心部分。下面将重点介绍这两个函数。

(1) Update 函数

MD5 类中有 4 个 Update 重载函数,其中 3 个公有函数是对外接口,在函数体中都调用了私有函数 Update 开启 MD5 摘要的计算过程。为了方便使用,3 个 Update 公有函数分别为字节流、字符串以及文件流提供了输入接口。虽然输入参数的类型不同,但是在这些接口函数中都会先将输入转化为标准字节流,再调用私有函数 Update。

由于 MD5 算法是以 64 字节(即 512bit)为单位进行计算的,私有函数 Update 首先需要对长度为 length 的字节流进行预处理,然后再调用 transform 函数对每一个 64 字节数据块进行计算。预处理并不是将字节流以 64 字节为单位简单地划分成若干个数据块,而是需要考虑前一次运算后缓存中是否保存着未被计算的字节。如果有,新的字节必须接在这些字节的后面进行填充,直到填满一个 64 字节数据块后才可以按上述方法继续划分下去;否则,直接以 64 字节为单位进行划分。当划分到最后剩余的字节数不足 64 字节时,将剩余的字节保存在缓存中,等待下一个字节流将数据块填满 64 字节后一起计算。私有函数 Update 的代码如下。

```
void MD5::Update(const BYTE* input,size_t length)
```

```
{
    DWORD i,index,partLen;

    //设置停止标识
    is_finished=false;
    //计算 buffer 已经存放的字节数
    index=(DWORD)((count[0]>>3) & 0x3f);

    //更新计数器 count,将新数据流的长度加上计数器原有的值
    if((count[0] +=((DWORD)length << 3))<((DWORD)length << 3))        //判断是否进位
        count[1]++;
    count[1] +=((DWORD)length>>29);

    //求出 buffer 中剩余的长度
    partLen=64-index;

    //将数据块逐块进行 MD5 运算
    if(length>=partLen)
    {
        memcpy(&buffer_block[index], input, partLen);
        Transform(buffer_block);

        for (i=partLen; i+63<length; i +=64)
            Transform(&input[i]);
        index=0;
    } else {
        i=0;
    }
    //将不足 64 字节的数据复制到 buffer_block 中
    memcpy(&buffer_block[index], &input[i], length-i);
}
```

如图 5-1 所示,Update 函数的流程可以分为下面 5 个步骤。

① 将标志 is_finished 设为 false,表示 MD5 运算正在进行。

② 将计数器 count 右移 3 位后再截取后 6 位,获得 buffer 中已经存放的字节数。

③ 更新计数器 count,将新数据流的长度加入计数器中。需要注意的是计数器 count 中保存的是 bit 数(等于字节数×8)。count[0]保存的是数值的低 32 位,count[1]保存的是数值的高 32 位。

④ 求出 buffer 中的剩余字节数 partLen。

⑤ 判断新数据流的长度 length 是否大于 partLen,如果 length 大于 partLen,将 parLen 长度的新数据拷贝至 buffer 中,使其填满 64 字节,然后调用 Transform 函数对 buffer 中的数据块进行 MD5 运算。接着利用循环将新数据流中的数据以 64 字节为单位,逐次进行 MD5 运算,直到剩余数据不足 64 字节为止,最后将新数据流中不足 64 字节的数据拷贝至 buffer 中。如果 length 不大于 partLen,则将新数据流的全部数据拷贝至 buffer 中即可。

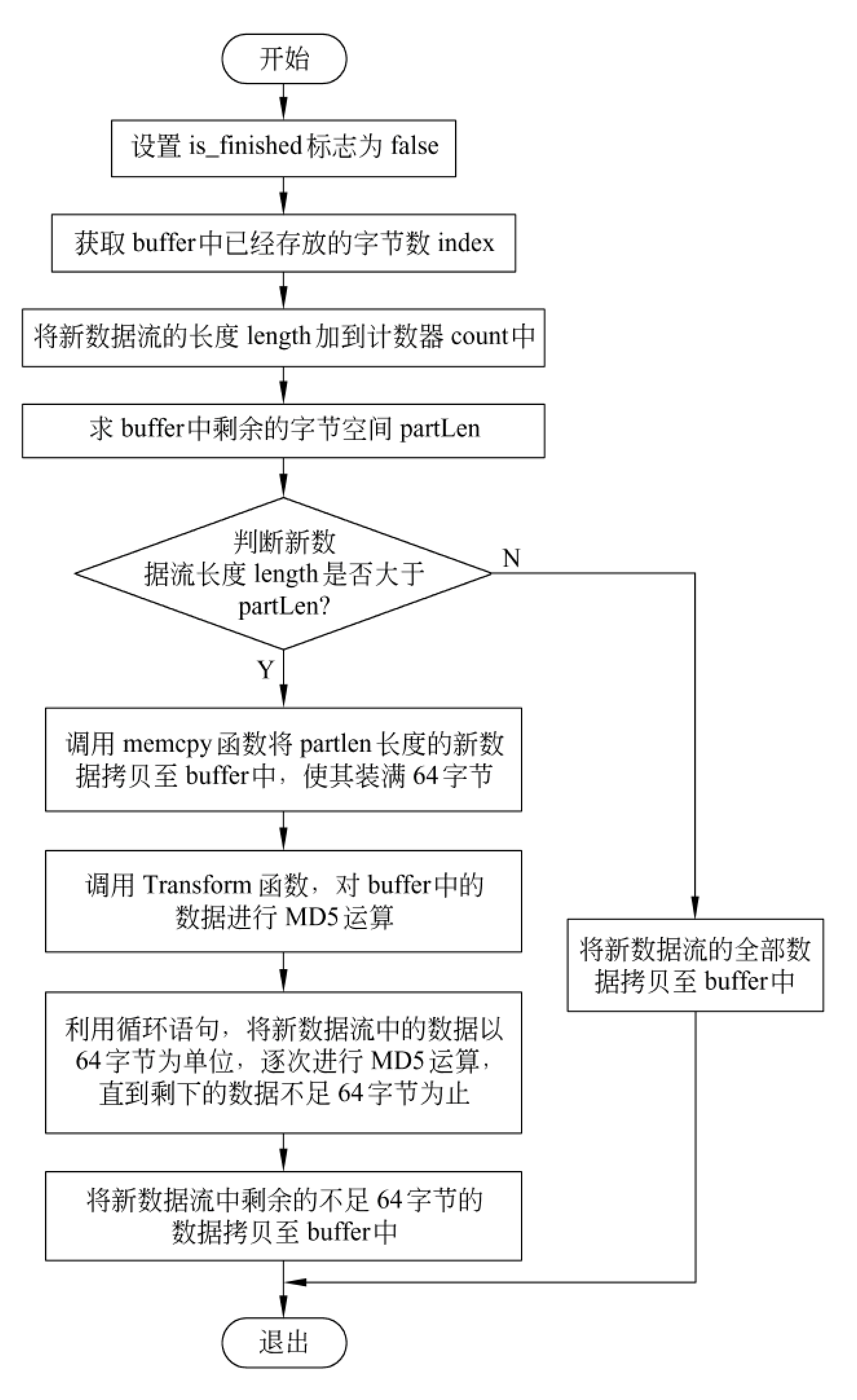


图 5-1 Update 函数流程图

(2) Transform 函数

在 MD5 类中，Transform 函数负责对 64 字节数据块进行 MD5 运算。在声明 Transform 函数之前，需要定义一些基本操作和基本运算。下面给出这些基本操作和运算的定义。

首先定义在 MD5 4 轮迭代计算中向量 ***A***、***B***、***C***、***D*** 循环左移的位数。例如，#define S11 7 表示第 1 轮计算中式子[ABCD，0，7，1]中循环左移的位数。如果需要查阅所有公式关于循环左移位数的定义，请参考 md5.cpp 文件中的代码。

其次定义 MD5 算法的 4 个基本函数，代码如下：

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))        //F函数
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))        //G函数
#define H(x, y, z) ((x) ^ (y) ^ (z))                   //H函数
```

```
#define I(x, y, z) ((y) ^ ((x) | (~z)))               //I 函数
```

然后定义 32 位双字的循环左移操作。如下所示，其中 x 表示一个 32 位双字，n 表示循环左移的位数。

```
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x)>> (32- (n))))
```

最后定义 4 轮计算中的 FF、GG、HH、II 函数。如下所示，其中 $\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$、$\boldsymbol{d}$ 表示计算向量，x 表示一个 32 位的子块，s 表示循环左移的位数，ac 表示弧度。

```
#define FF(a, b, c, d, x, s, ac) {
    (a) +=F ((b), (c), (d))+ (x)+ac;
    (a)=ROTATE_LEFT ((a), (s));
    (a) += (b);
}
#define GG(a, b, c, d, x, s, ac) {
    (a) +=G ((b), (c), (d))+ (x)+ac;
    (a)=ROTATE_LEFT ((a), (s));
    (a) += (b);
}
#define HH(a, b, c, d, x, s, ac) {
    (a) +=H ((b), (c), (d))+ (x)+ac;
    (a)=ROTATE_LEFT ((a), (s));
    (a) += (b);
}
#define II(a, b, c, d, x, s, ac) {
    (a) +=I ((b), (c), (d))+ (x)+ac;
    (a)=ROTATE_LEFT ((a), (s));
    (a) += (b);
}
```

根据上面定义的操作，Transform 函数实现了 MD5 摘要的计算过程。Transform 函数的定义如下所示。它的执行流程分为以下 4 个步骤：

① 首先将初始向量 state 的数值赋给变量 a、b、c、d。

② 调用 Decode 函数，将 64 字节的数据块划分为 16 个 32bit 大小的子分组。因为每一轮计算都是对 32bit 子分组进行操作，所以重新划分后可以方便后面的计算过程。

③ 依次调用函数 FF、GG、HH、II 展开 4 轮计算，其中每一轮计算包含 16 小步，每一步对一个 32bit 子分组进行运算。函数 FF、GG、HH、II 的前 4 个参数是变量 a、b、c、d 的不同排列，参数 $X[k]$表示对第 k 个子分组进行计算，S_{ij} 表示第 i 轮第 j 步计算循环左移的位数，最后一个常数 $T[i]$表示 $4294967296\times \text{abs}(\sin(i))$的整数部分。

④ 最后将变量 a、b、c、d 中的运算结果加到初始向量 state 中。

Transform 函数的实现代码如下。

```
void MD5::Transform(const BYTE block[64])
{
    DWORD a= state[0], b= state[1], c= state[2], d= state[3], x[16];
```

```
    Decode(block, x, 64);

    //第 1轮
    FF (a, b, c, d, x[ 0], S11, 0xd76aa478);          //1
    FF (d, a, b, c, x[ 1], S12, 0xe8c7b756);          //2
    …
    //第 2轮
    GG (a, b, c, d, x[ 1], S21, 0xf61e2562);          //17
    GG (d, a, b, c, x[ 6], S22, 0xc040b340);          //18
    …
    //第 3轮
    HH (a, b, c, d, x[ 5], S31, 0xfffa3942);          //33
    HH (d, a, b, c, x[ 8], S32, 0x8771f681);          //34
    …
    //第 4轮
    II (a, b, c, d, x[ 0], S41, 0xf4292244);          //49
    II (d, a, b, c, x[ 7], S42, 0x432aff97);          //50
    …

    state[0] +=a;
    state[1] +=b;
    state[2] +=c;
    state[3] +=d;
}
```

2. 文件完整性检验的设计与实现

文件完整性检验在 main 函数中实现。应用程序为用户提供了多个选项,不但可以在命令行下计算文件的 MD5 摘要,验证文件的完整性,还可以显示程序的帮助信息和 MD5 算法的测试信息。

在 main 函数中,程序通过区分参数 argv[1]的不同值来启动不同的工作流程。如果 argv[1]等于"-h",表示显示帮助信息;如果 argv[1]等于"-t",表示显示测试信息;如果 argv[1]等于"-c",表示计算被测文件的 MD5 摘要;如果 argv[1]等于"-v",表示根据手工输入的 MD5 摘要验证文件的完整性;如果 argv[1]等于"-h",表示根据.md5 文件中的摘要验证文件的完整性。

帮助信息可以协助用户快速地了解命令行输入格式。测试信息可以让用户验证 MD5 算法的正确性。用于测试的消息字符串都是 MD5 算法官方文档(RFC1321)中给出的例子,如果计算的结果相同,则说明程序的 MD5 运算过程正确无误。

程序提供了两种验证文件完整性的方式:一种是让用户手工输入被测文件的 MD5 摘要,然后调用 MD5 类的运算函数重新计算被测文件的 MD5 摘要,最后将两个摘要逐位进行比较,进而验证文件的完整性;另一种是从与被测文件对应的.md5 文件中读取 MD5 摘要,然后调用 MD5 类的运算函数重新计算被测文件的 MD5 摘要,最后将两个摘要逐位进

行比较，进而验证文件的完整性。

手工输入验证的代码如下。它分为以下 6 个步骤：

(1) 首先比较参数 argv[1]，判断是否通过手工输入进行验证。若是，则继续下面的步骤；否则，退出。

(2) 检测被测文件的路径是否存在，若存在，则继续下面的步骤；否则，退出。

(3) 输入被测文件的 MD5 摘要并保存在数组 InputMD5 中。

(4) 打开被测文件，读取被测文件的内容，并调用 Update 函数重新计算被测文件的 MD5 摘要。

(5) 调用 Tostring 函数将 MD5 摘要表示成十六进制字符串的形式。

(6) 最后调用 strcmp 函数判断两个摘要是否相同，若相同，则说明被测文件是完整的；否则，说明文件受到了破坏。

```
if (!strcmp(pValidate,argv[1]))                    //判断是否通过手工输入进行验证
{    …
    if (argv[2]==NULL)                             //判断是否输入了被测文件路径
    …
    //手动输入了被测文件的 MD5 摘要
    cout<<"Please input the MD5 value of file(\""<<argv[2]<<"\")..."<<endl;
    cin>>InputMD5;
    InputMD5[32]='\0';
    …
    //打开被测文件
    pFilePath=argv[2];
    ifstream File_2(pFilePath);
    …
    //读取文件内容并计算 MD5 摘要
    MD5 md5_obj3;
    md5_obj3.Reset();
    md5_obj3.Update(File_2);
    …
    //比较摘要,进行验证
    str=md5_obj3.Tostring();
    const char* pResult=str.c_str();
    if (strcmp(pResult,InputMD5))
    …
    else
    …
}
```

通过.md5 文件进行验证的代码如下。它与手工输入验证类似，也可分为以下 6 个步骤：

(1) 首先比较参数 argv[1]，判断是否通过.md5 文件进行验证。若是，则继续下面的步骤；否则，退出。

(2) 检测被测文件的路径和.md5 文件的路径是否存在，若存在，则继续下面的步骤；否

则，退出。

(3) 打开.md5文件，读取文件中的记录，调用strtok函数获得被测文件的MD5摘要。

(4) 打开被测文件，读取被测文件内容，并调用Update函数重新计算被测文件的MD5摘要。

(5) 调用Tostring函数将MD5摘要表示成十六进制字符串的形式。

(6) 最后调用strcmp函数判断两个摘要是否相同，若相同，则说明被测文件是完整的；否则，说明文件受到了破坏。

```
if (!strcmp(pFile,argv[1]))          //判断是否通过.md5文件进行验证
{    …
     //判断是否输入了被测文件和.md5文件路径
    if (argv[2]==NULL || argv[3]==NULL)
     …
     //打开.MD5文件
    pFilePath=argv[3];
    ifstream File_3(pFilePath);
     …
    //读取.MD5文件中的一行记录
    File_3.getline(Record,50);

    //以空格为标记,获得.MD5文件中的MD5值与对应文件名
    pMD5=strtok(Record,pS);
    pFileName=strtok(NULL,pS);
     …
    //打开被测文件
    pFilePath=argv[2];
    ifstream File_4(pFilePath);
     …
     //读取文件内容并计算MD5摘要
    MD5 md5_obj4;
    md5_obj4.Reset();
    md5_obj4.Update(File_4);
     …
    str=md5_obj4.Tostring();
    const char * pResult2=str.c_str();
     //比较摘要,进行验证
    if (strcmp(pResult2,pMD5))
       …
    else
       …
}
```

5.4 扩展与提高

5.4.1 MD5算法与Linux口令保护

除了验证文件完整性以外，MD5算法在其他方面也有着广泛的应用。Linux系统的用

户口令需要经过加密才能保存在配置文件中，而 MD5 算法则是加密用户口令的最常用算法。

最早，Linux 用户口令加密后保存在用户配置文件/etc/passwd 中。文件中的每一行对应一个用户，并用冒号(:)划分为 7 个字段。用户口令加密后就放在每一行的第 2 个字段里。一般情况下，/etc/passwd 文件允许所有用户读取，但只允许 root 用户写入。因此，恶意用户可以轻松地读取加密后的口令字符串，然后使用字典攻击等手段获得其他用户的口令。鉴于上述原因，后期的 Linux 版本专门设置一个用户影子口令文件/etc/shadow 用于保存用户口令，并且只允许 root 用户读取该文件。这样普通用户就无法读取加密后的口令文件，从而进一步提高了口令的安全性。

在/etc/passwd 文件中，每一行的第 2 个字段表示是否有加密口令。若为 x，则表示该账户设置了口令；否则，表示没有设置口令。以下面的/etc/passwd 文件为例，第 1 行记录表明 root 用户设置了口令。而真正的口令加密字符串则保存在/etc/shadow 文件中。

```
1 root:x:0:0:root:/root:/bin/bash
2 bin:x:1:1:bin:/bin:/sbin/nologin
…
```

与/etc/passwd 文件类似，/etc/shadow 文件的每一行对应一个用户，用冒号(:)划分了 9 个字段。用户加密后的口令就保存在第 2 个字段中。如下所示，root 用户的口令加密字符串是“1Q.DJ5Zss$P.WvrDK/ogM9w/KNlrEAR0”。该字符串是利用 MD5 算法生成的 128 位摘要。因为只有 root 用户才有权查看，所以有力地保障了 Linux 口令的安全性。

```
1 root:$ 1$ Q.DJ5Zss$ P.WvrDK/ogM9w/KNlrEAR0:13997:0:99999:7:::
2 bin:* :13948:0:99999:7:::
…
```

5.4.2　Linux 系统 GRUB 的 MD5 加密方法

1. GRUB 口令恢复的安全问题

在使用 Linux 过程中，用户经常会忘记自己之前设定的口令。鉴于这种情况，GRUB 为用户提供了一条恢复口令的捷径。GRUB 是一个引导装入器，负责装入内核并引导 Linux 系统。类似于在计算机上安装两个 windows 时出现的选单管理器 OS Loader，GRUB 可以让用户选择使用哪一个操作系统。与 Linux 之前的引导装入器 LILO 相比，GRUB 具有功能强大，引导过程灵活，安全性高等优点。

引导工具 GRUB 恢复口令的使用方法如下：

(1) 首先启动计算机，在 GRUB 界面时，选择需要进入的 Linux 系统，然后按 e 键。

(2) 选择以 kernel 开头的一行，按 e 键进入编辑模式。在此行的末尾按空格键后输入 single，代码如下所示。最后按回车键退出编辑。

```
kernel /boot/vmlinuz-2.4.18-14 single ro root=LABEL=/single
```

(3) 回到 GRUB 界面后，按 b 键来引导进入单用户模式。

(4) 进入单用户模式后，利用 password 命令设置新密码。

GRUB 虽然提供了一种恢复口令的方法，但是同时也带来了一个破解口令的漏洞。恶意用户完全可以利用上述方法获得 Linux 系统的用户口令。为了阻止未授权的用户登录启动引导程序，需要对 GRUB 设置密码。给 GRUB 设置密码一般有两种方式，一种是设置明文密码；另一种是利用 MD5 算法设置加密密码。

2. GRUB 设置明文密码

设置明文密码过程如下：在/etc 目录下打开配置文件 grub.conf，在“timeout=...”一行下面插入语句“password=****”。其中 * 代表明文密码。然后在“title...”一行下面插入语句“lock”。保存退出之后重启计算机。再次进入 GRUB 界面时，系统将提示需要按“p”键输入 GRUB 密码。

3. GRUB 设置 MD5 加密密码

将密码以明文形式保存在配置文件中是一种不安全的加密方法。如果利用 MD5 算法对 GRUB 密码进行加密，然后再将加密字符串保存在配置文件中，就可以进一步提高密码的安全性。MD5 加密密码的设置过程如图 5-2 所示。



图 5-2 GRUB 密码保护

(1) 首先生成 GRUB 的加密密码。在命令行输入 grub，进入 GRUB 界面。然后输入 md5crypt(或 password --md5)，再输入密码，就会生成一个 md5 加密字符串，代码如下所示。复制下该字符后，输入“quit”退出。

```
grub> md5crypt
Password:******
Encrypted: $1$mIpvT$y8iEtGvO7/IBRJdJhiD831
grub> quit
```

(2) 如下所示，在/etc 目录下打开配置文件 grub.conf，在“timeout=...”一行下面插入语句“password --md5 加密字符串”。然后在语句“title...”语句之后插入语句“lock”。保

存退出之后重启计算机。

```
[root@ localhost ~ ]# cd /etc
[root@ localhost etc]# vim grub.conf
…
default=0
timeout=10
password --md5$1$mLpvT$y8iEtGvO7/IBRJdJhiD831
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Fedora (2.6.23.1-42.fc8)
lock
        root (hd0,0)
        kernel /vmlinuz-2.6.23.1-42.fc8 ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        initrd /initrd-2.6.23.1-42.fc8.img
```

(3) 如图 5-2 所示，再次进入 GRUB 界面后，系统提示需要按“p”键输入 GRUB 密码才能引导启动。

经过上面的设置，未授权用户就不能轻易地引导系统，获得用户口令了。由此可见，MD5 算法在保护用户口令，维护 Linux 系统安全中扮演着重要的角色。

5.4.3　字典攻击与 MD5 变换算法

因为 MD5 算法的运算过程不可逆，所以不存在对任意一个 MD5 散列逆向计算出明文的破解机。目前，破解 MD5 的最有效方法之一就是字典攻击。所谓字典攻击，是指事先收集大量明文和对应的 MD5 散列，并把它们成对地保存在数据库中，然后在数据库中寻找与当前 MD5 散列对应的明文进行破解。当然如果数据库中没有相应的记录，破解工作就无法进行。

目前收集 MD5 字典的网站有很多，以下列出了 3 个有代表性的网站：

(1) http://md5.rednoize.com 网站采用搜索引擎的形式，支持明文字符串与 MD5 散列间的双向转换。目前拥有 1 963 442 条记录。

(2) http://www.neeao.com/md5/是一个中文网站，目前拥有近 17 亿条记录。

(3) http://www.xmd5.org/index_cn.htm 是一个多语言网站，目前拥有 2000 万条记录。

如果攻击者拥有数据量巨大的密码字典，并且建立了许多 MD5 原文/散列对照数据库，则能快速地找到常用明文字符串的 MD5 散列。因此字典攻击已经成为一种破解 MD5 散列的高效途径。然而，上述字典所使用的都是常规的 MD5 算法，即原文→MD5→密文。如果对 MD5 算法进行变换，就可以使这些 MD5 字典无所作为。

对 MD5 进行变换的方法有很多，最容易理解的就是对同一原文进行多次 MD5 运算。在下面给出的代码中，自定义了一个函数 md5_more，它有两个参数 data 和 times，前者表示需要加密的明文，后者表示重复加密的次数。由代码可见，在函数体中，调用了 times 次 md5 函数，因此将明文加密了 times 遍。这种变换也可以用递归的方法来实现。

```
function md5_more(data, times)
```

```
{  //循环使用 MD5
    for (i=0; i<times; i++) {
    data=md5(data);
    }
    return data;
}
```

另一种变换的主要思想是：首先经过一次 MD5 运算，得到一个由 32 个字符(用十六进制表示)组成的散列字符串，然后将该字符串分割为若干子串，再对每个子串进行 MD5 运算，最后将每个子串的 MD5 散列合并成一个字符串作为 MD5 算法的输入并计算出一个最终的散列。

在下面给出的代码中，函数 divide 首先对明文进行一次 MD5 运算，然后将得到的散列分为左、右两个字符串，各包含 16 个字符。分别对两个子串进行 MD5 运算后，再将计算结果合成一个 64 字节的字符串。最后用 MD5 算法将该字符串再计算一次，得到最终的结果。

```
//把密文分割成两段,每段 16个字符
function md5_divide(data)
{
    //先把明文加密成长度为 32字符的密文
    data=md5(data);
    //把密码分割成两段
    left=substr(data, 0, 16);
    right=substr(data, 16, 16);
    //分别加密后再合并
    data=md5(left).md5(right);
    //最后把长字符串再加密一次,成为 32字符散列
    return md5($ data);
}
```

除了上述两种变换方法之外，还有附加字符串干涉，字符串次序干涉以及大小写变换干涉等多种方法。可以说，MD5 变换算法以增加计算开销为代价，有效地抑制了字典攻击，提高了算法的安全性。但是，一旦攻击者获得了某个应用程序指定的 MD5 变换算法，则该变换算法就再也没有任何秘密可言。在这种情况下，变换后的算法只是比原算法消耗攻击者更多的计算资源。

第 6 章 基于Raw Socket的网络嗅探器程序

6.1 本章训练目的与要求

网络监控软件能够监测网络流量，发现网络中异常的数据流，有效地发现和防御网络攻击，是保证网络安全的重要工具和手段之一，也是网络安全技术人员必须掌握的重要技能之一。本章研究基于 Raw Socket 的网络嗅探器（Sniffer）系统设计与软件编程方法。

本章训练的主要目的是：

（1）理解 Sniffer 的基本工作原理与实现方法。

（2）掌握 Raw Socket 的基本工作原理。

（3）掌握 TCP/IP、ICMP 等协议及 socket 编程方法。

本章训练要求如下：

（1）利用原始套接字编写一个网络嗅探器捕获网络数据报。

（2）分析基本的数据报信息。

（3）实现简单的过滤器功能。

6.2 相关背景知识

6.2.1 原始套接字

Linux 系统套接字主要分为 3 类：TCP 套接字、UDP 套接字和原始套接字。TCP 套接字又称流式套接字，是建立在传输层 TCP 协议上的套接字；UDP 套接字又称数据报套接字，是建立在传输层 UDP 协议上的套接字；原始套接字是一种比较特殊的套接字，虽然创建原始套接字的方法和创建 TCP、UDP 套接字的方法基本相同，但是其功能和 TCP、UDP 套接字相比却存在很大的差异。

TCP/UDP 套接字只能接收和操作传输层或者传输层之上的数据报，因为当 IP 层把数据报往上传给传输层的时候，下层的数据报头部信息（如 IP 数据报头部和 Ethernet 帧头部）都已经去除了。而原始套接字可以直接对数据链路层的数据报进行操作。

1. 原始套接字特点

原始套接字有以下 5 个主要特点：

(1) 原始套接字可以读写 ICMP、IGMP 数据报。

对于 ICMP 和 IGMP 等封装在 IP 数据报中但是又在传输层之下的数据报，系统内核不管是否已经有注册的句柄来处理这些数据报，都会将这些数据报复制一份传递给协议类型匹配的原始套接字。通过这种方式，系统可以在用户空间中处理这些数据报，而不用内核来处理，从而可以减轻内核负担。

(2) 原始套接字可以读写部分的 IP 数据报。

通常这些数据报的协议域系统内核不能识别或者系统内核没有处理。对于内核不能识别其协议类型的数据报，内核首先会对其进行必要的校验，然后查看是否有匹配的原始套接字来处理，如果有匹配的原始套接字，内核会复制一份该数据报给匹配的原始套接字，否则，直接丢弃该 IP 数据报。

(3) 对于 TCP 包和 UDP 包，因为内核有相应的句柄对这些包进行处理，所以内核并不会将其传递给任何原始套接字。

如果想要通过原始套接字来捕获 TCP 和 UDP 包，在创建原始套接字的时候，必须将第 1 个参数设置为 PF_PACKET，第 3 个参数指定为 htons(ETH_P_IP)或者 htons(ETH_P_ALL)，这样原始套接字将直接通过数据链路层捕获数据报。本章实现的 Sniffer 程序就需要用这样的方式从链路层捕获所有流经本机网卡的数据报。

(4) 原始套接字可以构造自己的 IP 报头。

原始套接字可以构造自己的 IP 报头，这样就可以构造和发送特定的 IP 数据报。但是必须先通过 setsockopt()函数设置套接字 IP_HDRINCL 选项，即设置让程序自己填充 IP 报头，而不是由内核自动填充。

(5) 原始套接字需要超级用户权限才能创建。

2. 原始套接字的相关操作

(1) 创建原始套接字(代码如下)

```
int  rawsock;
rawsock= socket(domain, SOCK_RAW, protocol);
```

① 第 1 个参数 domain 表示地址族或者协议族，一般来说地址族用 AF_INET，协议族用 PF_INET。在功能上，PF_INET 和 AF_INET 是没有区别的。PF_INET 中的 PF_代表 protocol family，而 AF_代表 address family，在最初设计的时候是计划让一个 protocol family 包含多个 address family，但是这个计划并没有实现。在套接字的头文件中有 #define PF_INET AF_INET这样一个宏定义，因此这两个宏在数值上相等，在功能上也没有区别。如果需要原始套接字在链路层上捕获数据报，需要将该参数设置为 PF_PACKET。

② 第 2 个参数为套接字类型，原始套接字对应的类型为 SOCK_RAW。

③ 第 3 个参数 protocol 是一个常量定义，可以根据程序的需求选择相应的 protocol，形式如 IPPROTO_XXX。这些宏在头文件<netinet/in.h>中都有相应的定义。如果想要通过原始套接字来捕获 TCP 和 UDP 包，创建原始套接字的时候，要将该参数指定为 htons(ETH_P_IP)或者 htons(ETH_P_ALL)。

创建原始套接字需要超级用户权限，否则 socket 函数将不能成功创建原始套接字，返

回－1 值，同时会将 errno 置为 EACCES。

(2) 绑定和连接操作

通常情况下，原始套接字是不需要绑定操作的，但是也可以将原始套接字绑定在一个本地的地址上。原始套接字的绑定操作只是针对 IP 地址，而不会涉及端口。即原始套接字不会绑定在一个固定的端口上。调用 bind()函数之后，在接收数据报时，内核只会将目的 IP 地址为本机 IP 地址的数据报传递给该原始套接字。在原始套接字发送数据报时，数据报的源 IP 地址会自动设置为绑定的 IP 地址。如果没有调用 bind()函数，在接收数据报时，内核会把所有的数据报传递给该原始套接字，原始套接字发送数据报时，数据报的源 IP 地址会自动设置为发送接口的主 IP 地址。

原始套接字可以通过调用 connect()函数连接套接字。同样 connect()函数也只涉及 IP 地址，而不涉及端口号。调用 connect()函数，连接成功后，原始套接字就可以通过 write()和 send()函数发送数据报。

(3) 读写原始套接字

原始套接字的写操作通常使用 sendto()和 sendmsg()函数来完成。默认情况下，系统内核将自动填充 IP 头部。但是如果通过 setsockopt()函数设置了 IP_HDRINCL 选项，则必须在程序中手动填充 IP 头部，内核只负责填充后的 IP 头部的校验和。当数据报的长度大于链路中最大传输单元 MTU(Maximum Transmission Unit)时，系统内核会自动将该数据报进行分片。

原始套接字的读操作通常使用 recvfrom()和 recvmsg()函数。如果调用 connect()函数连接成功后就可以使用 recv()和 read()函数来接收数据报，同时内核也只会将源地址为 connect()函数连接的 IP 地址的数据报传递给这个原始套接字。

3. 数据报的处理

当系统收到一个数据报并且需要将其传递给原始套接字时，内核将检查所有进程的所有套接字，按照以下原则寻找匹配的原始套接字，然后将数据报拷贝给所有匹配的原始套接字。

(1) 原始套接字的协议域和数据报的协议域完全匹配并且为非 0 值，内核将数据报传递给该原始套接字。

(2) 如果原始套接字通过 bind()函数绑定到一个本地的 IP 地址上，则内核只会将目的 IP 地址和套接字绑定的 IP 地址匹配的数据报传递给该原始套接字。

(3) 如果原始套接字调用 connect()函数和远程的 IP 地址连接，则内核只将源 IP 地址和该远程 IP 地址匹配的数据报传递给该原始套接字。

如果一个原始套接字在创建时协议域为 0(即 socket()函数的第 3 个参数为 0)，并且没有调用 bind()和 connect()函数进行绑定和连接操作，则该原始套接字将接收所有内核传递给其他原始套接字的数据报。另外如前所述，原始套接字不能接收 TCP/UDP 包，只能接收 ICMP 包、IGMP 包以及一些协议域不被系统内核理解的数据报。所以如果想要通过原始套接字来捕获 TCP/UDP 包，在创建原始套接字的时候，必须将第 1 个参数设置为 PF_ PACKET，第 3 个参数设置为 htons(ETH_P_IP)或者 htons(ETH_P_ALL)。

6.2.2 TCP/IP 网络协议栈结构

图 6-1 给出了 TCP/IP 协议栈模型中常见的协议及其所在的位置。

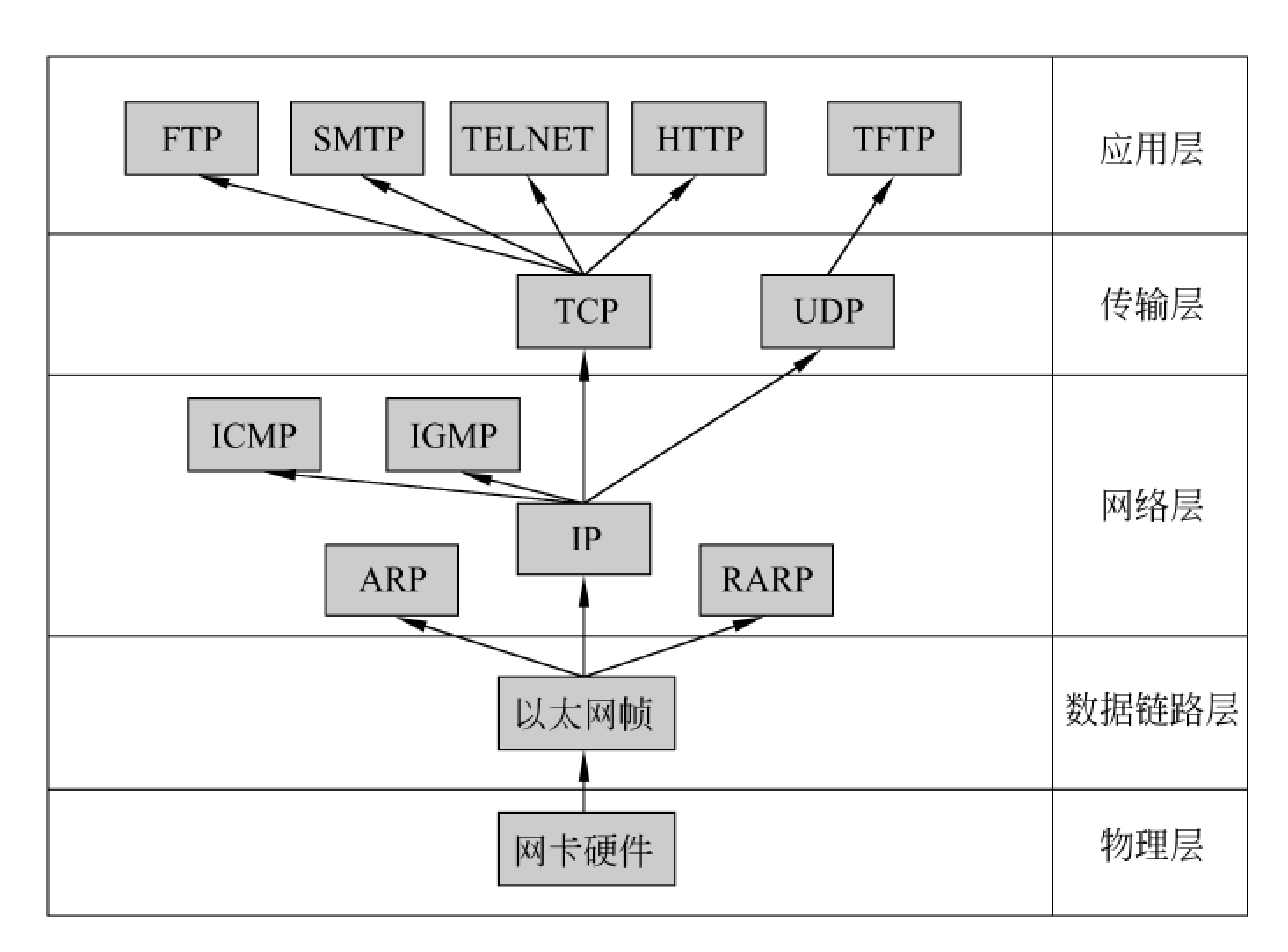


图 6-1 常见网络协议及其所在位置的示意图

6.2.3 数据的封装与解析

当应用程序通过协议栈向网络传送数据时，应用层的数据报要依次传递给传输层、互联层、数据链路层，最后通过物理层进入网络。在数据报经过每一层的时候，每一层都要在数据报中增加该层的头部(和尾部)信息，以此来实现层次控制，这个过程称为数据的封装。同样，当物理层收到数据报时，需要依次传递给数据链路层、网络层、传输层，最后送至应用层。在数据报经过每一层时，每一层都需要对对应于该层的头部(和尾部)信息进行解析，最终从报文中解析出应用层数据后交给应用程序进行处理。该过程称为数据拆包或解析。图 6-2 给出了数据的封装与解析过程的示意图。

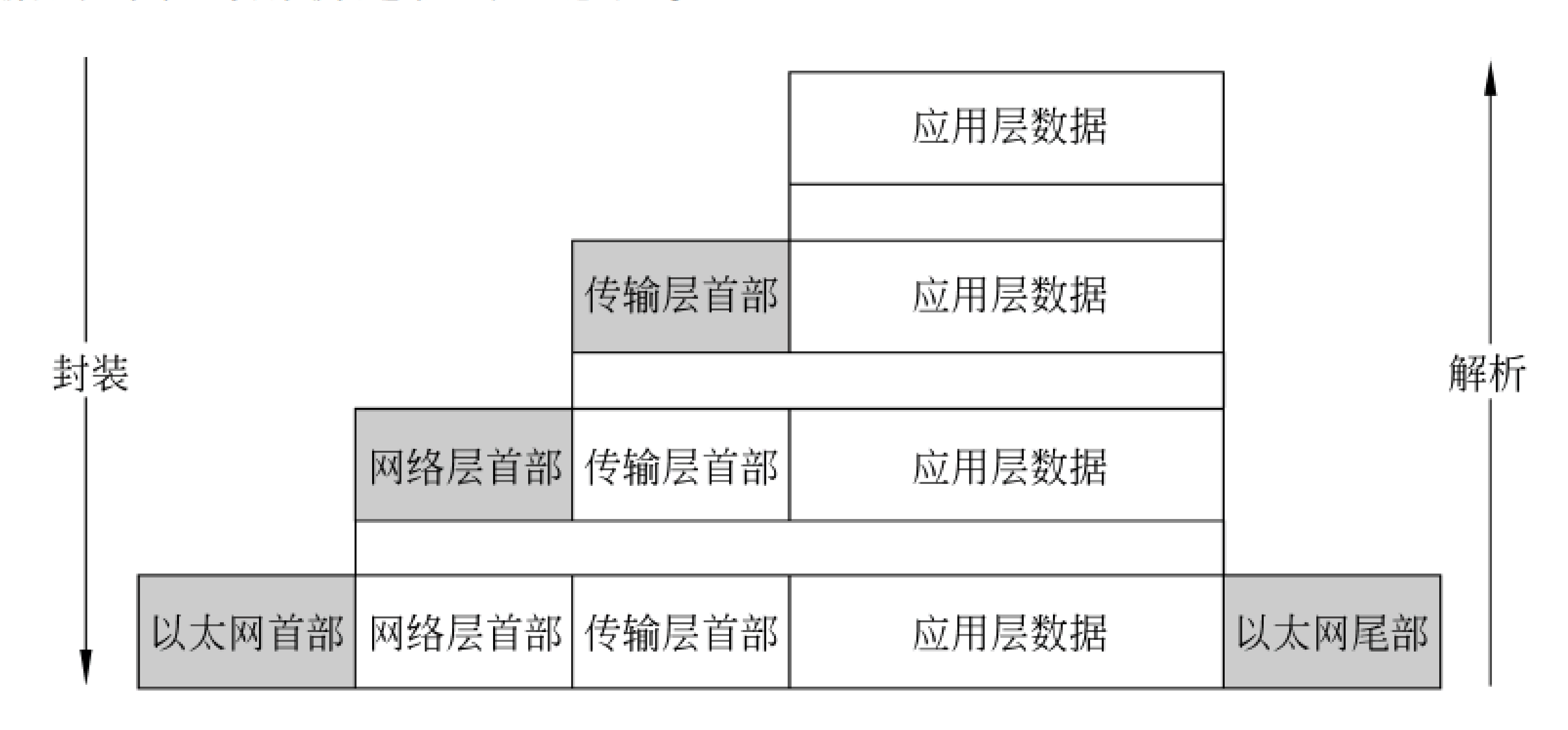


图 6-2 数据的封装与解析过程示意图

本章要实现的 Sniffer 程序就是利用原始套接字捕获所有流经本机网卡的数据报并对其进行解析。按照图 6-2 的解析过程，由底层往上逐层解析，最终将每层解析的结果显示

出来。

6.3　实例编程练习

6.3.1　编程练习要求

根据前面章节介绍的有关原始套接字的知识，利用原始套接字编写一个网络嗅探器捕获网络数据报并且分析基本的数据报信息，例如 IP 地址、端口号、协议类型、物理地址等。另外要求实现简单的过滤器功能，能够捕获指定的数据报，例如捕获指定 IP 地址、指定协议的数据报等。

6.3.2　编程训练设计与分析

1. 程序整体框架

现代操作系统通常都提供了对底层网络数据报捕获的机制。虽然不同的操作系统实现的底层数据报捕获机制可能不一样，但是从功能上来说却是大同小异。数据报从网络传递到主机应用程序的路径依次为网卡、设备驱动层、数据链路层、网络层、传输层，最后到达应用层程序。本章关注的数据报捕获机制主要是在数据链路层。捕获的数据报是内核对该数据报的一份拷贝，这样的数据报捕获机制不会影响操作系统对数据报进行网络协议栈的处理操作。整个程序的主体框架如图 6-3 所示。

本程序主体结构主要分成 3 个部分，由下而上，分别是数据捕获模块、数据报过滤模块和协议解析模块。首先由数据捕获模块从网络中捕获数据报，然后在数据报过滤模块中按照设定的过滤规则将数据报进行过滤，最后再将过滤后的数据报通过协议解析模块解析显示出来。

本实验要求用原始套接字来捕获数据报。原始套接字不仅可以捕获数据报，还可以发送指定的数据报。但是在本实验的程序中只需要用到原始套接字捕获数据报的功能。通过原始套接字捕获数据报的流程如图 6-4 所示。

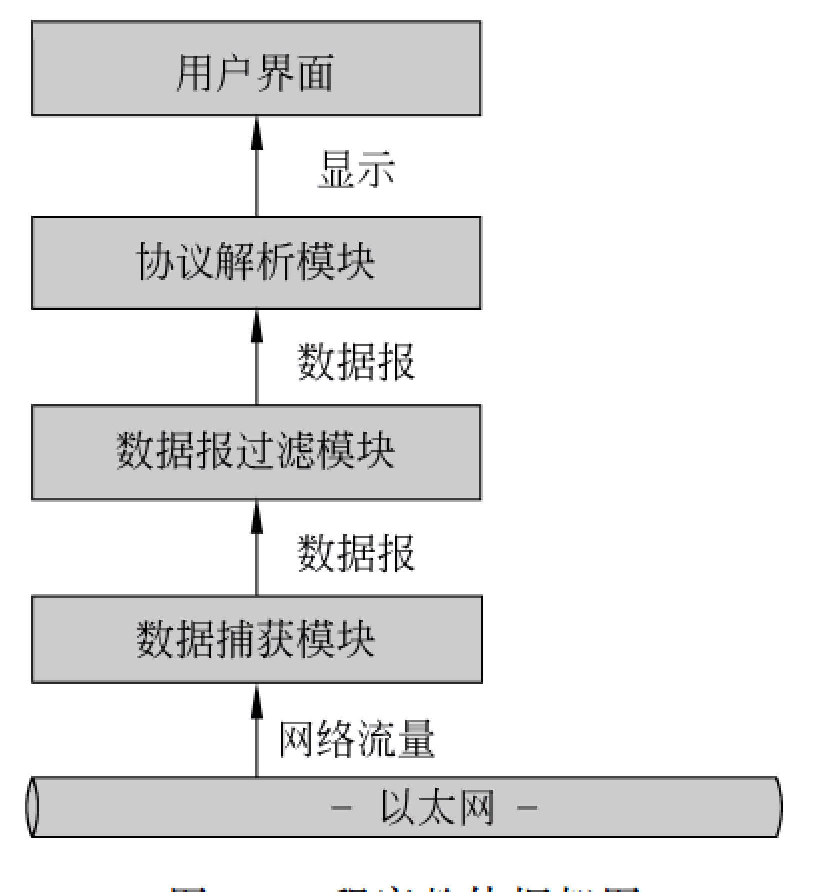


图 6-3　程序整体框架图

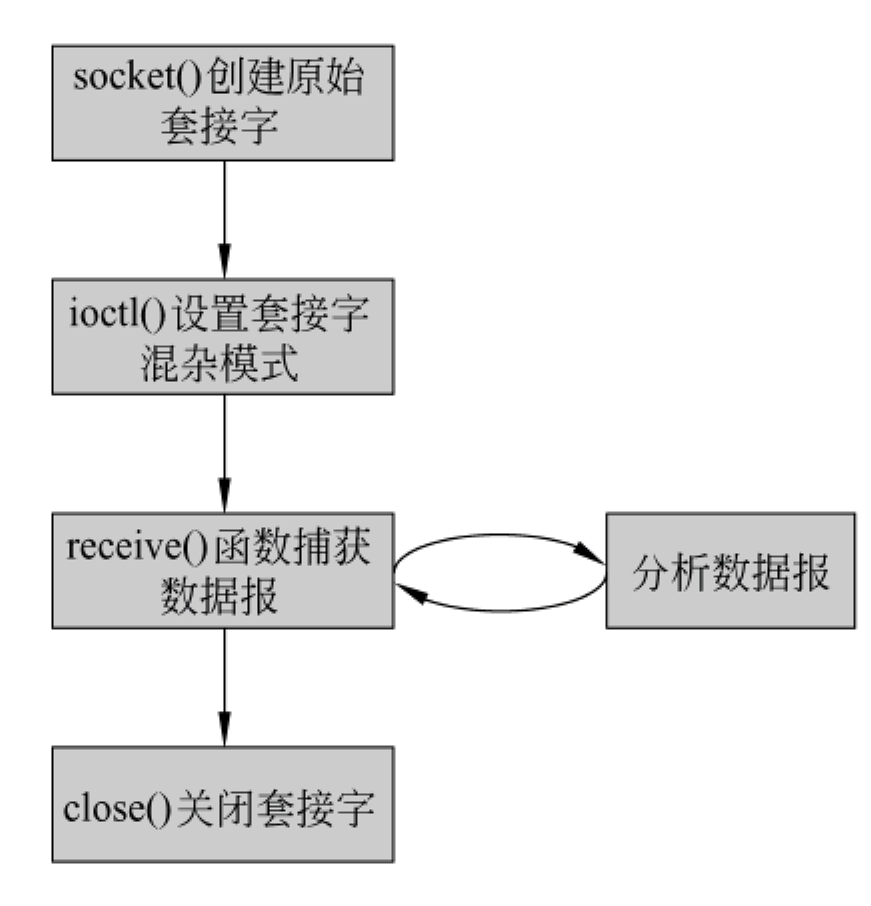


图 6-4　原始套接字捕获数据报流程图

2. 程序核心代码

程序定义了两个类，一个是 rawsocket 类，另一个是 rawsocsniffer 类。

rawsocket 类对原始套接字的一些设置和操作进行封装，避开了很多烦琐的原始套接字操作，使程序更加直观，也使得对原始套接字的操作和扩充变得更加方便。

rawsocsniffer 类对整个嗅探器的主要功能进行封装，包括创建原始套接字，设置混杂模式，捕获数据报并且设置过滤，对数据报进行分析等。

下面给出各个模块的核心代码。为了使程序的流程更加清晰，代码没有做很严格的错误处理。在实际应用中应该增加更为严格的错误处理，以此来保证程序的健壮性。

(1) rawsocket 类

rawsocket 类的定义如下。

```
class rawsocket
{
    private:
    int sockfd;
    public:
    rawsocket(const int protocol);
    ~rawsocket();
    //set the promiscuous mode.
    bool dopromisc(char* nif);
    //capture packets.
    int receive(char* recvbuf,int buflen,struct sockaddr_in* from,int* addrlen);
};
```

rawsocket 类维护 1 个数据成员 sockfd，即原始套接字句柄。维护 4 个成员函数，构造函数根据传入的参数构造原始套接字，在析构函数中关闭该原始套接字。dopromisc()函数对 ioctl()函数进行了封装，用于设置网卡混杂模式。receive()函数对原始套接字的 recvfrom()函数进行了封装，用于捕获数据报。对原始套接字的其他设置和操作都可以封装在该类中。由于本章的程序只需要用到上面这些操作，所以其他的操作没有封装在该类中，有兴趣的读者可以对其进行扩充。

① 构造函数

构造函数根据参数传入的协议类型创建原始套接字(代码如下)。

```
rawsocket::rawsocket(const int protocol)
{
    sockfd=socket(PF_PACKET,SOCK_RAW,protocol);
    if(sockfd<0)
    {
        perror("socket error:");
    }
}
```

② dopromisc()函数

dopromisc()函数设置网卡为混杂模式。该函数以网卡的名字为参数。要对指定的网

卡设置混杂模式，只要以该网卡名字为参数调用 dopromisc()函数即可。如果需要恢复网卡模式只需将代码“ifr. ifr_flags |= IFF_PROMISC”中的“|=”改为“&=”即可(代码如下)。

```
bool rawsocket::dopromisc(char * nif)
{
    struct ifreq ifr;
    strncpy(ifr.ifr_name, nif,strlen(nif)+1);
    if((ioctl(sockfd, SIOCGIFFLAGS, &ifr)==-1))
    {
        perror("ioctlread:");
        return false;
    }
    ifr.ifr_flags |=IFF_PROMISC;
    if(ioctl(sockfd, SIOCSIFFLAGS, &ifr)==-1 )
    {
         perror("ioctlset:");
        return false;
    }
    return true;
}
```

③ receive()函数

receive()函数对原始套接字的 recvfrom()函数进行了封装，实现了对原始套接字的 recvfrom()函数更方便地调用。当程序成功地捕获了一个数据报后，recvfrom()函数的返回值为接收到的数据报的长度(代码如下)。

```
int rawsocket::receive(char * recvbuf,int buflen, struct sockaddr_in * from,int * addrlen)
{
    int recvlen;
    recvlen=recvfrom(sockfd,recvbuf,buflen,0,(struct sockaddr * )from,(socklen_t * )addrlen);
    recvbuf[recvlen]='\0';
    return recvlen;
}
```

(2) rawsocsniffer 类

rawsocket 类的定义如下。

```
class rawsocsniffer:public rawsocket
{
    private:
    filter simfilter;
    char * packet;
    const int max_packet_len;
    public:
    rawsocsniffer(int protocol);
```

```
    ~rawsocsniffer();
    bool init();
    void setfilter(filter myfilter);
    bool testbit(const unsigned int p, int k);
    void setbit(unsigned int &p,int k);
    void sniffer();
    void analyze();
    void ParseRARPPacket();
    void ParseARPPacket();
    void ParseIPPacket();
    void ParseTCPPacket();
    void ParseUDPPacket();
    void ParseICMPPacket();
    void print_hw_addr(const unsigned char * ptr);
    void print_ip_addr(const unsigned long ip);
};
```

rawsocsniffer 类对嗅探器的主要功能进行了封装。该类继承自 rawsocket 类。rawsocsniffer 类维护 3 个数据成员：一个 filter 类型的数据结构 simfilter 用于设置过滤条件，一个 char 型指针用于存储数据报，另一个 int 型数据用于记录最大数据报的长度。

① 设置过滤

过滤功能主要是通过 simfilter 数据成员来实现的，simfilter 的结构如下。

```
typedef struct filter
{
    unsigned long sip;
    unsigned long dip;
    unsigned int protocol;
} filter;
```

该程序实现了针对源 IP、目的 IP 和协议类型进行过滤的功能。filter 结构维护了 3 个变量：源 IP、目的 IP 和协议类型。程序初始化时会对 rawsocsniffer 类的 simfilter 结构进行设置来设定过滤条件。例如，如果只要求捕获来自 IP 地址为“192.168.0.45”的数据报，则将 simfilter 结构中的 sip 设置为 192.168.0.45 即可。同样如果要对目的 IP 进行过滤，则需要设置 simfilter 的 dip 为指定 IP。对协议类型进行过滤是通过对 protocol 变量进行操作来实现的。protocol 变量中的每一位对应一种协议类型。如果某位为 1 则表示接收该位所对应协议的数据报，为 0 则表示不接收。protocol 变量中哪一位对应哪种协议类型可以按照自定义的规则来设置。在本程序中，arp 协议对应 protocol 的第 1 位，tcp 协议对应第 2 位，udp 协议对应第 3 位，icmp 协议对应第 4 位。在按协议类型分析数据报之前，先查看 protocol 变量中该协议对应的位是否为 1，如果为 1 则分析该数据报，否则丢弃。例如，如果 protocol 变量为 0x0002，二进制为(0000000000000010)，其中第 2 位为 1，其他位全为 0，则表示程序只分析 tcp 包。如果 protocol 变量为 0x0005，二进制为(0000000000000101)，其中第 1 位和第 3 位为 1，其余位都为 0，则表示程序只分析 udp 包和 arp 包。如果 protocol 等于 0x0000，所有位都为 0，即没有设置过滤，则程序将分析所有捕获的数据报。读者可以

对该过滤机制进行扩充，以实现更多的过滤功能。rawsocsniffer 类有以下 3 个函数对 simfilter 变量进行维护。

a．setfilter()函数

setfilter()函数根据传入的参数设置 simfilter(代码如下)。

```
void rawsocsniffer::setfilter(filter myfilter)
{
    simfilter.protocol=myfilter.protocol;
    simfilter.sip=myfilter.sip;
    simfilter.dip=myfilter.dip;
}
```

b．testbit()函数

testbit()函数用于测试某一无符号整型变量的指定位是否为 1(代码如下)。

```
bool rawsocsniffer::testbit(const unsigned int p,int k)
{
    if((p>>(k-1))&0x0001)
        return true;
    else
        return false;
}
```

c．setbit()函数

setbit()函数用于将某一无符号整型变量的指定位置 1(代码如下)。

```
void rawsocsniffer::setbit(unsigned int &p,int k)
{
    p=(p)|((0x0001)<<(k-1));
}
```

② 数据报捕获

sniffer()函数用于启动数据报的捕获过程，该函数通过循环调用基类 rawsocket 的 receive 函数来捕获数据报，receive 函数的返回值为捕获到的数据报长度，所以当返回值大于 0 时表示捕获到数据报，然后调用 analyze()函数对捕获到的数据报进行处理(代码如下)。

```
void rawsocsniffer::sniffer()
{
    struct sockaddr_in from;
    int sockaddr_len=sizeof(struct sockaddr_in);
    int recvlen=0;
    while(1)
    {
        recvlen=receive(packet,max_packet_len,&from,&sockaddr_len);
        if(recvlen>0)
        {
```

```
                analyze();
            }
            else
            {
                continue;
            }
        }
    }
```

rawsocsniffer 类维护了一个数据成员 packet 来临时存储捕获到的数据报。packet 是一个 char 型指针，rawsocsniffer 类的构造函数给该指针分配了一块大小为 max_packet_len 的内存空间。在本程序中 max_packet_len 的值设置为 2048。

3. 数据报头部格式

本程序要求分析一些常见的协议，如 ARP 协议、IP 协议、TCP 协议、UDP 协议和 ICMP 协议。各种协议的数据报头部结构定义如下。

(1) 以太网帧头部

以太网帧头部结构定义如下：

```
typedef struct ether_header_t{
    BYTE des_hw_addr[6];            //目的 MAC 地址
    BYTE src_hw_addr[6];            //源 MAC 地址
    WORD frametype;                 //数据长度或类型
} ether_header_t;
```

(2) IP 包头部

IP 包头部结构定义如下：

```
typedef struct ip_header_t{
    BYTE hlen_ver;                  //头部长度和版本信息
    BYTE tos;                       //8 位服务类型
    WORD total_len;                 //16 位总长度
    WORD id;                        //16 位标识符
    WORD flag;                      //3 位标志+13 位片偏移
    BYTE ttl;                       //8 位生存时间
    BYTE protocol;                  //8 位上层协议号
    WORD checksum;                  //16 位校验和
    DWORD src_ip;                   //32 位源 IP 地址
    DWORD des_ip;                   //32 位目的 IP 地址
} ip_header_t;
```

(3) ARP 包头部

ARP 包头部结构定义如下：

```
typedef struct arp_header_t{
    WORD hw_type;                       //16 位硬件类型
```

```
    WORD prot_type;              //16 位协议类型
    BYTE hw_addr_len;            //8 位硬件地址长度
    BYTE prot_addr_len;          //8 位协议地址长度
    WORD flag;                   //16 位操作码
    BYTE send_hw_addr[6];        //源 Ethernet 网地址
    DWORD send_prot_addr;        //源 IP 地址
    BYTE des_hw_addr[6];         //目的 Ethernet 网地址
    DWORD des_prot_addr;         //目的 IP 地址
} arp_header_t;
```

(4) TCP 包头部

TCP 包头部结构定义如下：

```
typedef struct tcp_header_t{
    WORD src_port;               //源端口
    WORD des_port;               //目的端口
    DWORD seq;                   //seq 号
    DWORD ack;                   //ack 号
    BYTE len_res;                //头长度
    BYTE flag;                   //标志字段
    WORD window;                 //窗口大小
    WORD checksum;               //校验和
    WORD urp;                    //紧急指针
} tcp_header_t;
```

(5) UDP 包头部

UDP 包头部结构定义如下：

```
typedef struct udp_header_t{
    WORD src_port;               //源端口
    WORD des_port;               //目的端口
    WORD len;                    //数据报总长度
    WORD checksum;               //校验和
} udp_header_t;
```

(6) ICMP 包头部

ICMP 包头部结构定义如下：

```
typedef struct icmp_header_t{
    BYTE type;                   //8 位类型
    BYTE code;                   //8 位代码
    WORD checksum;               //16 位校验和
    WORD id;                     //16 位标识符
    WORD seq;                    //16 位序列号
} icmp_header_t;
```

4. 数据报解析

数据报的解析过程就是对捕获的数据报按照数据链路层(MAC 协议)、网络层(IP、

ARP/RARP 协议)、传输层(TCP、UDP、ICMP 协议)和应用层(HTTP 协议)的层次结构自底向上进行解析,分析各个协议的字段,最后将解析结果显示输出。数据报的解析过程如图 6-5 所示。

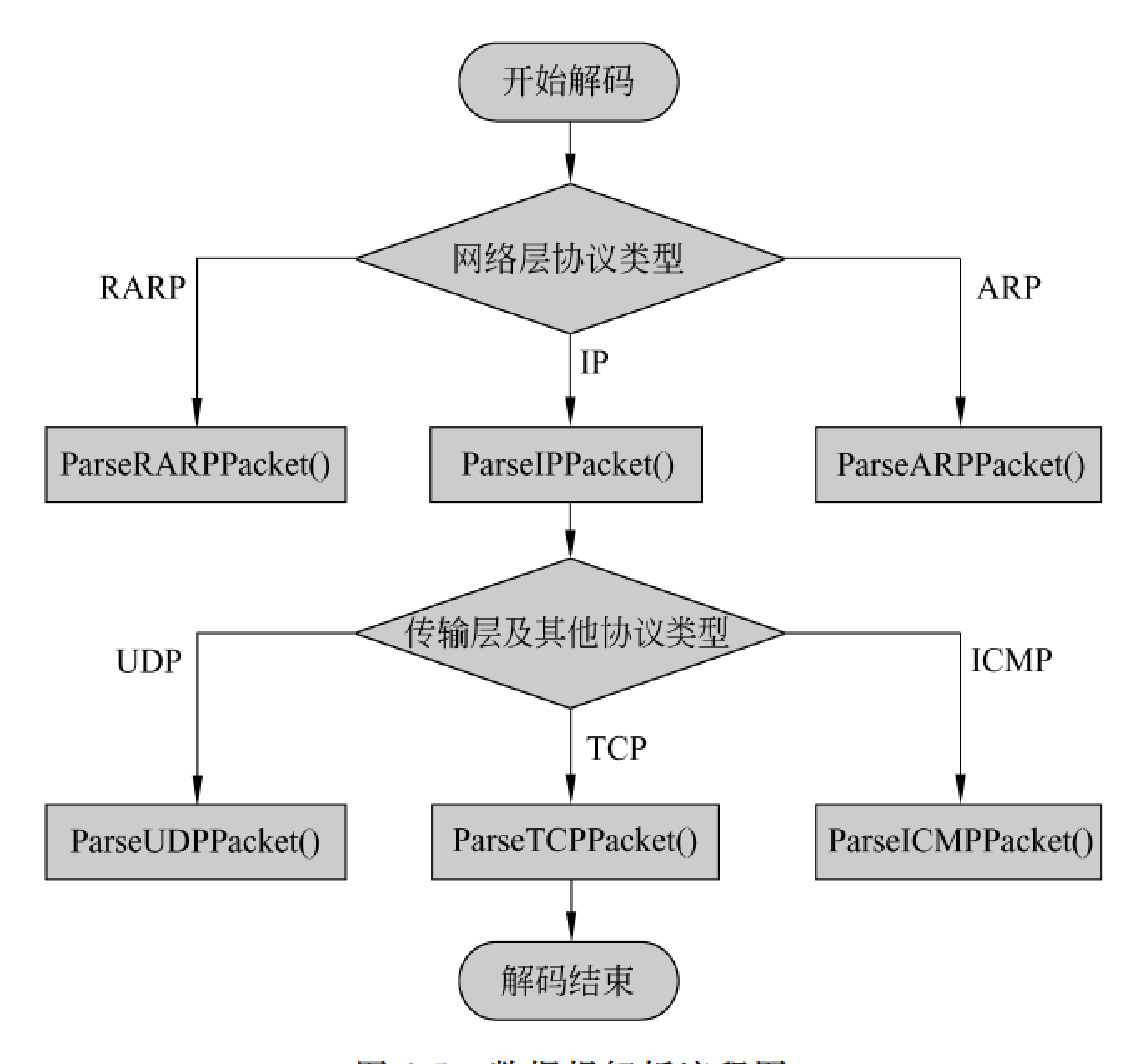


图 6-5　数据报解析流程图

程序每捕获到一个数据报就调用 analyze()函数对该数据报进行解析。analyze()函数根据以太帧的类型调用相应的上层解析函数对数据报进行解析。帧类型 0x0800 表示上层为 IP 包,0x0806 表示上层为 ARP 包,0x0835 表示上层为 RARP 包。调用上层解析函数前先根据过滤器的设置判断是否要对该协议的数据报进行解析(代码如下所示)。

```
void rawsocsniffer::analyze()
{
    ether_header_t * etherpacket= (ether_header_t * )packet;
    if(simfilter.protocol==0)
    simfilter.protocol=0xff;
    switch (ntohs(etherpacket->frametype))
    {
    case 0x0800:
        if(((simfilter.protocol)>>1))
        {
            cout<<"\n\n/ * ------------ ip packet----------------- * /"<<endl;
            ParseIPPacket();
        }
        break;
    case 0x0806:
        if(testbit(simfilter.protocol,1))
        {
```

```
                cout<<"\n\n/*------------arp packet-----------------*/"<<endl;
                ParseARPPacket();
            }
            break;
        case 0x0835:
            if(testbit(simfilter.protocol,5))
            {
                cout<<"\n\n/*------------RARP packet----------------*/"<<endl;
                ParseRARPPacket();
            }
            break;
        default:
            cout<<"\n\n/*------------Unknown packet-----------------*/"<<endl;
            cout<<"Unknown ethernet frametype!"<<endl;
            break;
    }
}
```

(1) 解析 ARP 包

程序解析了 ARP 包的几个主要字段，包括硬件地址长度、协议地址长度、协议类型、操作类型以及源 IP 地址、源 MAC 地址、目的 IP 地址及目的 MAC 地址。其中操作类型为 0x0001 表示 ARP 请求，0x0002 表示 ARP 应答(代码如下)。

```
void rawsocsniffer::arppacket_analyze()
{
    arp_packet_t* arppacket=(arp_packet_t*)packet;
    print_hw_addr(arppacket->arpheader.des_hw_addr);
    print_hw_addr(arppacket->arpheader.send_hw_addr);
    cout<<endl;
    print_ip_addr(arppacket->arpheader.send_prot_addr);
    print_ip_addr(arppacket->arpheader.des_prot_addr);
}
```

(2) 解析 IP 包

程序首先判断过滤条件，根据过滤器的源 IP 和目的 IP 对数据报进行过滤。然后再根据 IP 层协议域字段的值来调用对应的上层协议解析函数对数据报进行解析。其中协议域字段值为 1 表示上层为 ICMP 包，6 表示 TCP 包，17 表示 UDP 包(代码如下)。

```
void rawsocsniffer::ParseIPPacket()
{
    ip_packet_t* ippacket=(ip_packet_t*)packet;
    cout<<"ipheader.protocol:"<<int(ippacket->ipheader.protocol)<<endl;
    if(simfilter.sip!=0)
    {
        if(simfilter.sip!=(ippacket->ipheader.src_ip))
            return;
```

```
    }
    if(simfilter.dip!=0)
    {
        if(simfilter.dip!=(ippacket->ipheader.des_ip))
        return;
    }
    switch (int(ippacket->ipheader.protocol))
    {
    case 1:
        if(testbit(simfilter.protocol,4))
        {
            cout<<"Received an ICMP packet"<<endl;
            ParseICMPPacket();
        }
        break;
    case 6:
        if(testbit(simfilter.protocol,2))
        {
            cout<<"Received an TCP packet"<<endl;
            ParseTCPPacket();
        }
        break;
    case 17:
        if(testbit(simfilter.protocol,3))
        {
            cout<<"Received an UDP packet"<<endl;
            ParseUDPPacket();
        }
        break;
//省略针对其他协议的分析
    }
}
```

(3) 解析 ICMP 包

程序解析了 ICMP 包的类型、编码、标示符和序列号字段(代码如下)。

```
void rawsocsniffer::icmppacket_analyze()
{
    icmp_packet_t* icmppacket=(icmp_packet_t*)packet;
    cout<<setw(20)<<"MAC address: from";
    print_hw_addr(icmppacket->etherheader.src_hw_addr);
    cout<<"to";
    print_hw_addr(icmppacket->etherheader.des_hw_addr);
    cout<<endl<<setw(20)<<"IP address: from";
    print_ip_addr(icmppacket->ipheader.src_ip);
    cout<<"to";
```

```
    print_ip_addr(icmppacket->ipheader.des_ip);
    cout<<endl;
    cout<<setw(12)<<"icmp type:"<<int(icmppacket->icmpheader.type)<<"icmp code:"
    <<int(icmppacket->icmpheader.code)<<endl;
    cout<<setw(12)<<"icmp id:"<<ntohs(icmppacket->icmpheader.id)<<" icmp seq:"
    <<ntohs(icmppacket->icmpheader.seq)<<endl;
}
```

(4) 解析 TCP 包

程序解析了 TCP 包的源端口、目的端口、序列号和 ACK 等字段(代码如下)。

```
void rawsocsniffer::tcppacket_analyze()
{
    tcp_packet_t* tcppacket=(tcp_packet_t*)packet;
    cout<<setw(20)<<"MAC address: from";
    print_hw_addr(tcppacket->etherheader.src_hw_addr);
    cout<<"to";
    print_hw_addr(tcppacket->etherheader.des_hw_addr);
    cout<<endl<<setw(20)<<"IP address: from";
    print_ip_addr(tcppacket->ipheader.src_ip);
    cout<<"to";
    print_ip_addr(tcppacket->ipheader.des_ip);
    cout<<endl;
    cout<<setw(10)<<"srcport:"<<ntohs(tcppacket->tcpheader.src_port)<<"desport:"
    <<ntohs(tcppacket->tcpheader.des_port)<<endl;
    cout<<"seq:"<<ntohl(tcppacket->tcpheader.seq)<<"ack:"<<ntohl(tcppacket->tcpheader.ack)<<
   endl;
}
```

(5) 解析 UDP 包

程序解析了 UDP 包的源端口、目的端口和数据报长度等字段(代码如下)。

```
void rawsocsniffer::udppacket_analyze()
{
    udp_packet_t* udppacket=(udp_packet_t*)packet;
    cout<<setw(20)<<"MAC address: from";
    print_hw_addr(udppacket->etherheader.src_hw_addr);
    cout<<"to";
    print_hw_addr(udppacket->etherheader.des_hw_addr);
    cout<<endl<<setw(20)<<"IP address: from";
    print_ip_addr(udppacket->ipheader.src_ip);
    cout<<"to";
    print_ip_addr(udppacket->ipheader.des_ip);
    cout<<endl;
    cout<<setw(10)<<"srcport:"<<ntohs(udppacket->udpheader.src_port)<<"desport: "
    <<ntohs(udppacket->udpheader.des_port)\
```

```
        <<"length:"<<ntohs(udppacket->udpheader.len)<<endl;
    }
```

6.4 扩展与提高

6.4.1 使用 libpcap 捕获数据报

1. libpcap 介绍

对数据报的捕获也可以借助 Linux 下的 libpcap 开发包来实现。该开发包是由 Berkeley 大学的 Van Jacobson、Craig Leres 和 Steven McCanne 合作开发的，它提供了丰富的 API 函数，可以帮助程序员快速地开发数据报捕获软件。libpcap 的官方网址是：www.tcpdump.org。读者可以在此网站上下载该开发包。该开发包对捕获网络数据报的功能进行了封装，使用起来非常方便。并且支持自定义过滤规则，只捕获用户感兴趣的数据报。其过滤规则非常详细，可以进行各种复杂组合的过滤。由于它的过滤模块是在内核层次中实现的，所以效率非常高。很多著名的软件如抓包工具 tcpdump 和网络入侵检测系统 snort 都是基于 libpcap 开发的。

2. libpcap 的功能

libpcap 的功能可以归纳为以下几点：

（1）捕获数据报

捕获数据报是 libpcap 最基本也是最强大的功能。使用 libpcap 可以方便、高效地捕获网络数据报。

（2）过滤数据报

libpcap 提供了强大的过滤机制，在内核层中对数据报进行过滤，效率非常高，而且其过滤规则非常详细，可以进行各种复杂组合，实现强大的过滤功能。

（3）分析数据报

libpcap 在捕获数据报时提供了一些辅助信息，比如捕获时间、数据报长度等信息。可以帮助开发者更好地分析数据报。

（4）存储数据报

libpcap 提供了将数据报存储到本地的功能，从网上捕获数据报后，可以先将其存储在本地电脑上，以后再对其进行分析。libpcap 提供了一种机制对离线的数据报进行分析，即从 libpcap 保存在本地的文件中读取数据报进行分析。

3. libpcap 进行数据报捕获和分析的步骤

使用 libpcap 进行数据报捕获和分析的步骤如下：

（1）使用函数 pcap_findalldevs()获取设备列表。

（2）使用函数 pcap_lookupnet()获取网络地址和子网掩码。

（3）使用函数 pcap_open_live()打开指定的设备。

（4）使用函数 pcap_compile()编译过滤规则。

(5) 使用函数 pcap_setfilter()设置过滤规则。

(6) 使用函数 pcap_loop()来循环捕获数据报,并在该函数中调用数据报分析模块解析数据报。

(7) 使用函数 pcap_close()关闭设备句柄。

使用 libpcap 捕获并分析数据报的流程如图 6-6 所示。

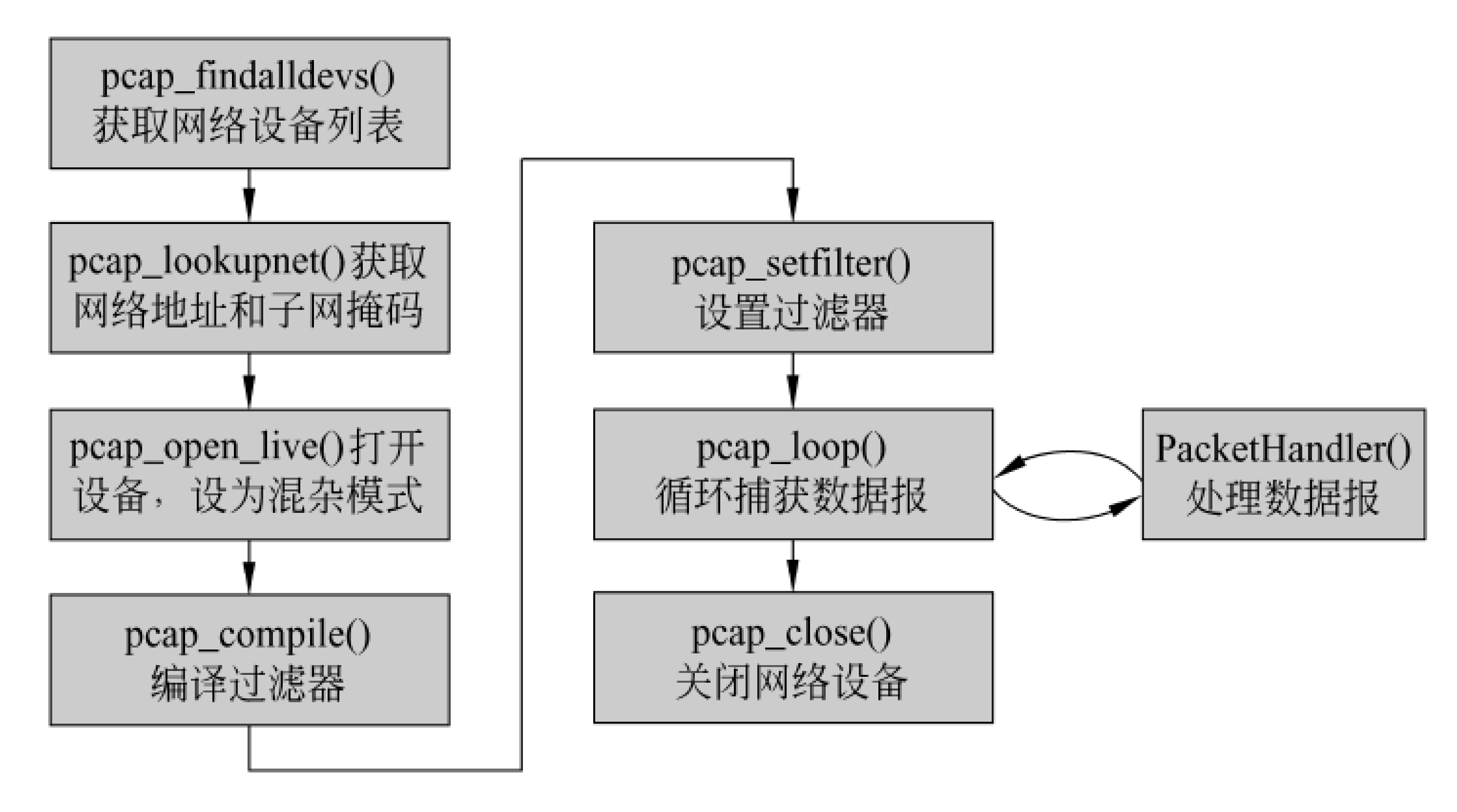


图 6-6　数据报捕获流程图

4. 使用 libpcap 捕获数据报的关键代码

(1) 获取并且输出网络设备列表信息

首先通过 pcap_findalldevs()函数获取本机所有网络设备的列表,并且将其存储在 device 变量中。然后通过 ifprint()函数将找到的设备信息打印出来(代码如下)。

```
if(pcap_findalldevs(&devices,errbuf)==-1)
{
    printf("Error in pcap_findalldevs():%s \n",errbuf);
    return-1;
}
else
{
    printf("Find the following devices on your machine: \n");
    ifprint(devices,devname);
}
```

(2) 获取网络地址和子网掩码

其中参数 devname 为网络设备的名字,获取的网络地址存储在变量 net_ip 中,子网掩码存储在变量 net_mask 中(代码如下)。

```
if(pcap_lookupnet(devname,&net_ip,&net_mask,errbuf)==-1)
{
    printf("Error in the pcap_lookupnet: %s\n",errbuf);
    goto error;
}
```

(3) 打开指定设备

其中参数 devname 为要打开的设备名字(代码如下)。

```
if((dev_handle_pcap=pcap_open_live(devname,BUFSIZ,1,0,errbuf))==NULL)
{
    printf("Error in the pcap_open_live!\n");
    goto error;
}
```

(4) 编译和设置过滤规则

其中参数 bpf_filter 的数据类型为 struct bpf_program,参数 bpf_filter_string 是设置的过滤字符串。libpcap 中支持多种类型的过滤,功能非常强大。包括基于 IP 地址、MAC 地址、端口号等类型的过滤,还可以通过一些关键字设置过滤表达式,组合出各种过滤类型。设置过滤器最主要的工作在于如何根据规则构造过滤表达式(代码如下)。

```
if((pcap_compile(dev_handle_pcap,&bpf_filter,bpf_filter_string,0,net_ip))==-1)
{
    printf("Error in the pcap_compile!\n");
    goto error;
}
else
{
    if((pcap_setfilter(dev_handle_pcap,&bpf_filter))==-1)
    {
        printf("Error in the pcap_setfilter!\n");
        goto error;
    }
}
```

(5) 过滤表达式

过滤表达式的语法主要有以下几点。

① 表达式支持逻辑操作符,可使用关键字 and、or 和 not 对子表达式进行组合,同时支持使用小括号。

② 基于协议的过滤要使用协议限定符,协议限定符可以为 ip、arp、rarp、tcp 和 udp 等。

③ 基于 MAC 地址的过滤要使用限定符 ether(代表 Ethernet 网地址)。当该 MAC 地址仅作为源地址时过滤表达式为 ether src mac_addr,仅作为目的地址时表达式为 ether dst mac_addr,既作为源地址又作为目的地址时表达式为 ether host mac_addr。此外应注意 mac_addr 应遵循由冒号分隔的十六进制格式,如 00:E0:4C:E0:38:88,否则编译过滤器时会出错。

④ 基于 IP 地址的过滤应使用限定符 host(代表主机地址)。当该 IP 地址仅作为源地址时过滤表达式为 src host ip_addr,仅作为目的地址时表达式为 dst host ip_addr,既作为源地址又作为目的地址时表达式为 host ip_addr。

⑤ 基于端口的过滤应使用限定符 port。例如仅接收 80 端口的数据报则表达式为

port 80。

设置过滤字符串的示例如下。

例 1：只捕获 arp 或 icmp 包

过滤表达式为：arp or (ip and icmp)，或者简写为：arp or icmp

例 2：捕获以 192.168.1.27 为源或目的地址的端口为 80 的 tcp 包

过滤表达式为：(ip and tcp) and (host 192.168.1.27) and (port 80)

例 3：捕获主机 192.168.1.27 与 192.168.1.22 之间传递的所有 udp 包

过滤表达式为：(ip and udp) and ((src host 192.168.1.27 and dst host 192.168.1.22)or (dst host 192.168.1.27 and src host 192.168.1.22))

例 4：捕获从 mac 地址 00-13-D3-A1-D2-F6 发送至 00-50-56-C0-00-01 所有的 arp 包

过滤表达式为：arp and (ether src 00:13:D3:A1:D2:F6 and ether dst 00:50:56:C0:00:01)

(6) 捕获数据报(代码如下)

```
pcap_loop(dev_handle_pcap,- 1,Packethandler,NULL);
```

其中第 1 个参数是已经打开的设备接口句柄，第 3 个参数是数据报处理函数，该函数有固定的格式，其定义如下。

```
void Packethandler(u_char * argument,const struct pcap_pkthdr * packet_header,const u_char * packet_
content)
```

pcap_loop 函数捕获到一个数据报后会自动调用 Packethandler 函数对数据报进行处理。并且自动将该数据报的信息作为参数传递给 Packethandler 函数。可以在 Packethandler 函数中增加对数据报进行解析的代码，其中第 2 个参数 packet_header 为 libpcap 在捕获数据报时增加的辅助头部信息，包括时间戳、数据报长度等，第 3 个参数 packet_content 即为捕获到的数据报。至此，就可以按照前面介绍的数据报的解析过程对数据报进行解析。解析过程同图 6-5 一致。

(7) 关闭设备

程序结束时注意要关闭设备，释放资源(代码如下)。

```
pcap_close(dev_handle_pcap);
```

5. 注意事项

(1) 程序在最后编译时必须加上编译选项-lpcap，表示需要用到 libpcap 的库函数，编译才能通过。

(2) libpcap 只能捕获数据报而不能发送数据报(windows 下的 winpcap 可以发送数据报)，如果要想进行数据报的发送操作则需要用到 Linux 下的另一个开发包 libnet，通过该开发包提供的一些接口函数来填充和发送数据报。

6.4.2 使用 tcpdump 捕获数据报

1. tcpdump 介绍

tcpdump 是一种功能很强的网络数据截取分析工具。它支持针对网络层、协议、主机、网络或端口的过滤，支持基于正则表达式的过滤方式，具有强大的功能和灵活的截取策略。tcpdump 主要用于网络的分析、维护、统计、检测，如定位网络瓶颈、统计网络流量使用情况等。更重要的是 tcpdump 是开源项目，提供了源代码，公开了接口，因此具备很强的可扩展性，对于网络维护和管理者来说是非常有用的工具。大多数的 Linux 操作系统都将 tcpdump 集成在内。因为 tcpdump 需要将网卡设置为混杂模式，所以需要 root 权限才能执行 tcpdump 命令。

2. tcpdump 命令

该命令使用的语法如下。

```
tcpdump  [-adeflnNOpqRStuvxX] [-c count] [-C file_size] [-F file]
         [-i interface] [-m module] [-r file] [-s snaplen] [-T type] [-w file]
         [-E algo:secret] [expression]
```

可以通过对 tcpdump 选项、表达式进行组合，以及对参数进行设定，从大量的网络数据中过滤出有用的信息来分析网络问题。

3. tcpdump 选项(option)

tcpdump 可用的选项如表 6-1 所示。

表 6-1 tcpdump 可用的选项

选 项	含 义
-a	把网络地址和广播地址转换成名字
-dd	将匹配数据报代码以 c 程序格式输出
-ddd	将匹配的数据报代码以十进制形式输出
-e	输出数据链路层的头部信息
-f	将外部的 Internet 地址以数字的形式打印出来
-l	对标准输出进行缓冲，可以在捕捉数据的同时查看数据
-n	不把网络地址转换成主机名
-N	不输出主机名中的域名部分，例如“tcpdump. org”只输出“tcpdump”
-O	不运行数据报匹配模板的优化器
-p	不将网络接口设置成混杂模式
-q	快速输出，只输出较少的协议信息

续表

选　项	含　　义
-S	将 tcp 的序列号以绝对值的形式输出，而不是相对值
-t	不在输出的每一行打印时间戳
-u	输出未解码的 NFS 句柄
-v	输出比较详细的信息，例如在 ip 包中可以包括 ttl 和服务类型的信息
-vv	输出详细的报文信息
-vvv	输出更为详细的报文信息
-c count	指定监听数据报数量，当收到指定的包的数目后退出 tcpdump
-C file_size	限定数据报写入文件的大小
-F file	从指定的文件中读取过滤正则表达式，忽略命令行中的表达式
-i interface	指定监听网络接口
-m module	打开指定的 SMI MIB 组件
-r file	从指定的文件中读取数据报（这些数据报一般通过-w 选项产生）；
-s snaplen	从每个数据报中读取最开始的 snaplen 个字节，而不是默认的 68 个字节；将截取的数据报直接解释为指定类型的报文
-T type	把捕获的数据报解析成指定的 type。目前已知的类型有：rpc、rtp、rtcp、vat 和 wb
-w file	将捕获的数据报写入文件，不分析和打印数据报
-E algo:secret	用 algo:secret 解密 IPsec ESP 数据报

4．tcpdump 表达式（expression）

tcpdump 利用正则表达式来过滤数据报。如果没有指定表达式，则捕获全部数据报，否则，只捕获满足表达式为真的数据报。表达式有 3 类常用的关键字：type、dir 和 proto。

（1）type 是指定类型的关键字，包括 host、net 和 port。缺省为 host。

例如：tcpdump host 192.168.1.27 表示捕获 IP 为 192.168.1.27 的主机收发的所有数据报。

（2）dir 是指定数据报传输方向的关键字，包括 src、dst、src or dst 和 src and dst。缺省为 src or dest。

例如：tcpdump dst net 192.168.1.28 表示捕获目标网络地址为 192.168.1.28 的所有数据报。

（3）proto 是指定协议的关键字，包括 ether、fddi、tr、ip、ip6、arp、rarp、decnet、tcp 和 udp。

例如：tcpdump udp 表示捕获所有 udp 协议的数据报。

（4）其他重要的关键字还有 gateway、broadcast、less、greater 和 3 种逻辑运算符（取非运算符 “not”或者“!”；与运算符“and”或者“&&”；或运算是“or”或者“‖”）将这些关键字灵活地组合就能构造出各种复杂的过滤条件。

5. 简单示例

(1) 捕获 arp 和 udp 包：

```
tcpdump arp or udp
```

(2) 捕获除了主机名为 alice 之外的所有 IP 包：

```
tcpdump ip host not alice
```

(3) 捕获主机 IP 地址为 192.168.1.102 接收和发出的 telnet 数据报：

```
tcpdump tcp and host 192.168.1.102 and port 23
```

(4) 捕获源 IP 地址为 192.168.1.1，目的 IP 地址为 192.168.1.102，端口为 80 的数据报：

```
tcpdump tcp and src host 192.168.1.1 and dst host 192.168.1.102 and port 80
```

(5) 捕获主机 IP 地址为 192.168.1.1 和 192.168.1.102 之间传递的所有 tcp 包：

```
tcpdump ip and tcp and\ (\ (src host 192.168.1.1 and dst host 192.168.1.102\) or\ (dst host 192.168.1.1 and src
host 192.168.1.102\)\)
```

6. 注意事项

在表达式中用到括号时，一定要在括号前加反斜杠。以上示例只是利用正则表达式组合出的一些简单的过滤条件，读者完全可以根据自己的需求利用正则表达式组合出更复杂的过滤条件。

第 7 章 基于OpenSSL的安全Web服务器程序

7.1 本章训练目的与要求

Web 服务使用的传输协议是 HTTP 协议。HTTP 采用明文传输，网络传输中的重要数据有被第三方截获的危险。安全超文本传输协议(HTTP over SSL，HTTPS)用于保护敏感数据在 Web 系统中的传输安全。HTTPS 通过安全套接字协议层(Secure Socket Layer，SSL)加密 HTTP 数据，并可以与 HTTP 数据共存。因此，研究基于 OpenSSL 的安全 Web 服务器软件的设计与编程方法，对于提高 Web 系统的安全性有着重要的意义。

本章训练的主要目的是：

(1) 理解 HTTPS 协议与 SSL 协议的基本工作原理。

(2) 掌握使用 OpenSSL 编程的方法。

(3) 掌握安全 Web 系统设计的基本设计与编程方法。

本章训练的要求是：

(1) 在 Linux 平台上利用 OpenSSL 库，编写一个 Web Server 程序。

(2) Server 程序要能够并发处理多个请求，要求至少能支持 HTTPS 协议下最基本的 Get 命令，可以增强 Web Server 的功能，如支持 Head、Post 以及 Delete 命令等。

(3) 编写必要的客户端测试程序，用于发送 HTTPS 请求并显示返回结果。

7.2 相关背景知识

7.2.1 SSL 协议介绍

SSL 是 Netscape 公司于 1996 年提出的安全通信协议，它为网络应用层的通信提供了认证、数据保密和数据完整性服务。设计 SSL 的主要目的是为网络环境中两个通信应用进程之间提供一个安全通道。

图 7-1 给出了 SSL 协议栈的结构示意图。SSL 借助 TCP 协议来提供端到端的安全服务，SSL 并不是一个单独的协议，而是两层结构的协议集合，上层包括 SSL 握手协议、SSL 修改密文规约协议和 SSL 警告协议，下层包括 SSL 记录协议。

1. SSL 记录协议

SSL 记录协议为通信提供机密性和完整性保护，图 7-2 给出了该协议的工作流程，具体

SSL 握手协议	SSL 修改密文规约协议	SSL 警告协议	HTTP
SSL 记录协议			
TCP			
IP			

图 7-1　SSL 协议栈结构示意图

工作过程如下：

(1) 接收到应用层数据以后，SSL 记录协议首先对其进行分组，分组后数据块的长度不超过 2^{14}(16384)字节。

(2) 对数据块进行压缩，压缩过程中不能出现信息的丢失，同时增加的长度不能超过 1024B(压缩处理是可选的，现有的 SSL3.0 和 TLS1.0 都没有指定压缩算法)。

(3) 在压缩后的数据上计算消息认证码 MAC，并把 MAC 附加在数据块之后。

(4) 对添加 MAC 后的数据块进行加密，加密可以采用流加密或块加密的方式。

(5) 为加密后的数据添加 SSL 记录协议的头部。

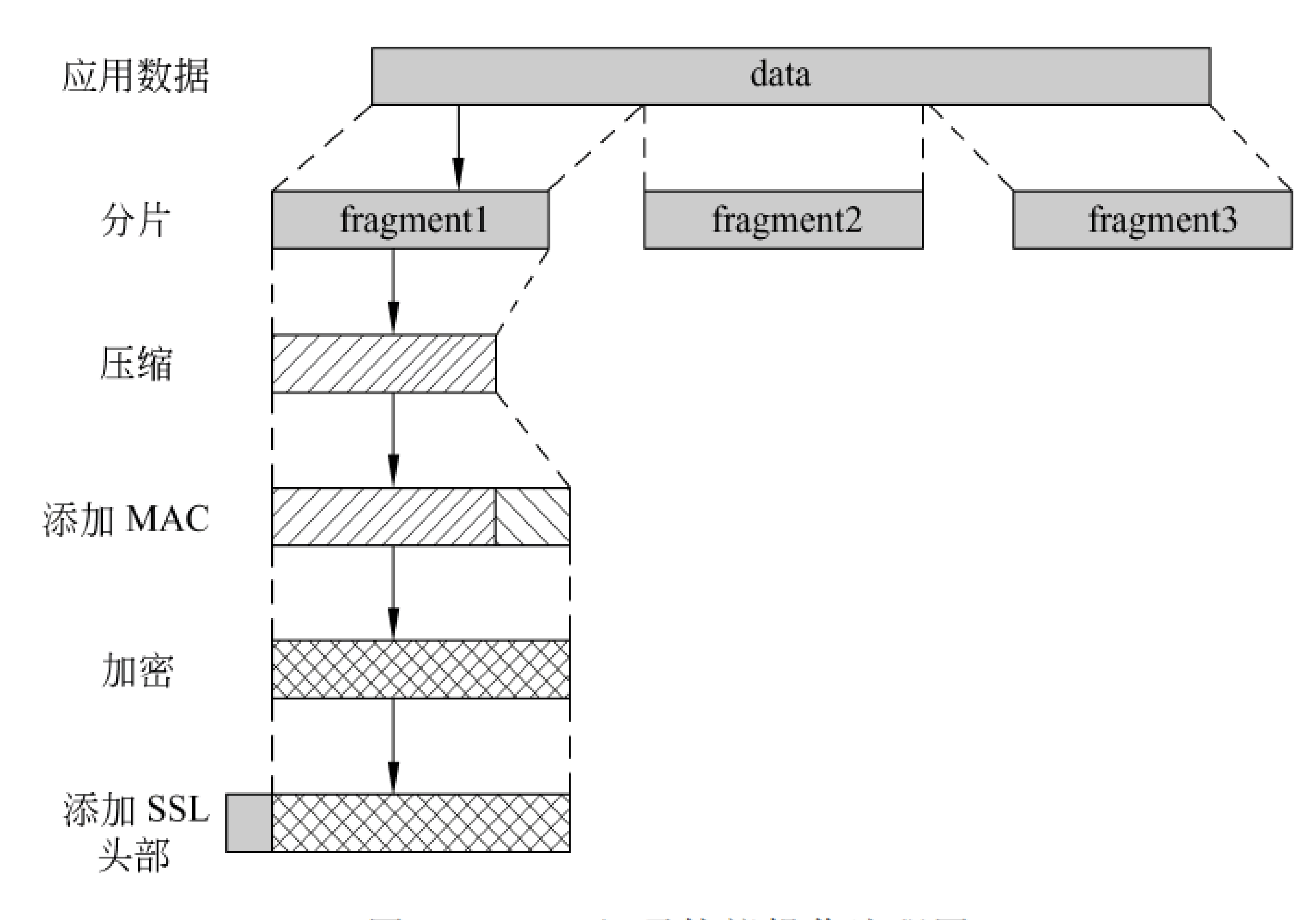


图 7-2　SSL 记录协议操作流程图

2. SSL 握手协议

SSL 握手协议是对 SSL 会话状态进行维护，为通信双方建立安全的传输通道，它是 SSL 协议中最复杂的部分。

当 SSL 客户端和服务器首次通信时，双方通过握手协议，协商通信协议的版本号、选择密码算法、互相认证身份(可选)，并使用公钥加密技术通过一系列的交换消息在客户端和服务器之间生成共享的秘密。双方根据这个秘密信息产生数据加密算法和 Hash 算法的参数等。

握手协议在应用层数据传输之前进行，包含一系列服务器与客户端的报文交换，这些报文都含有 3 个字段，即消息类型(1B)、消息长度(3B)和消息内容(不少于 1B)。

图 7-3 描述了客户端和服务器建立连接时的报文交换过程，整个交换过程可以分为以下 4 个阶段。

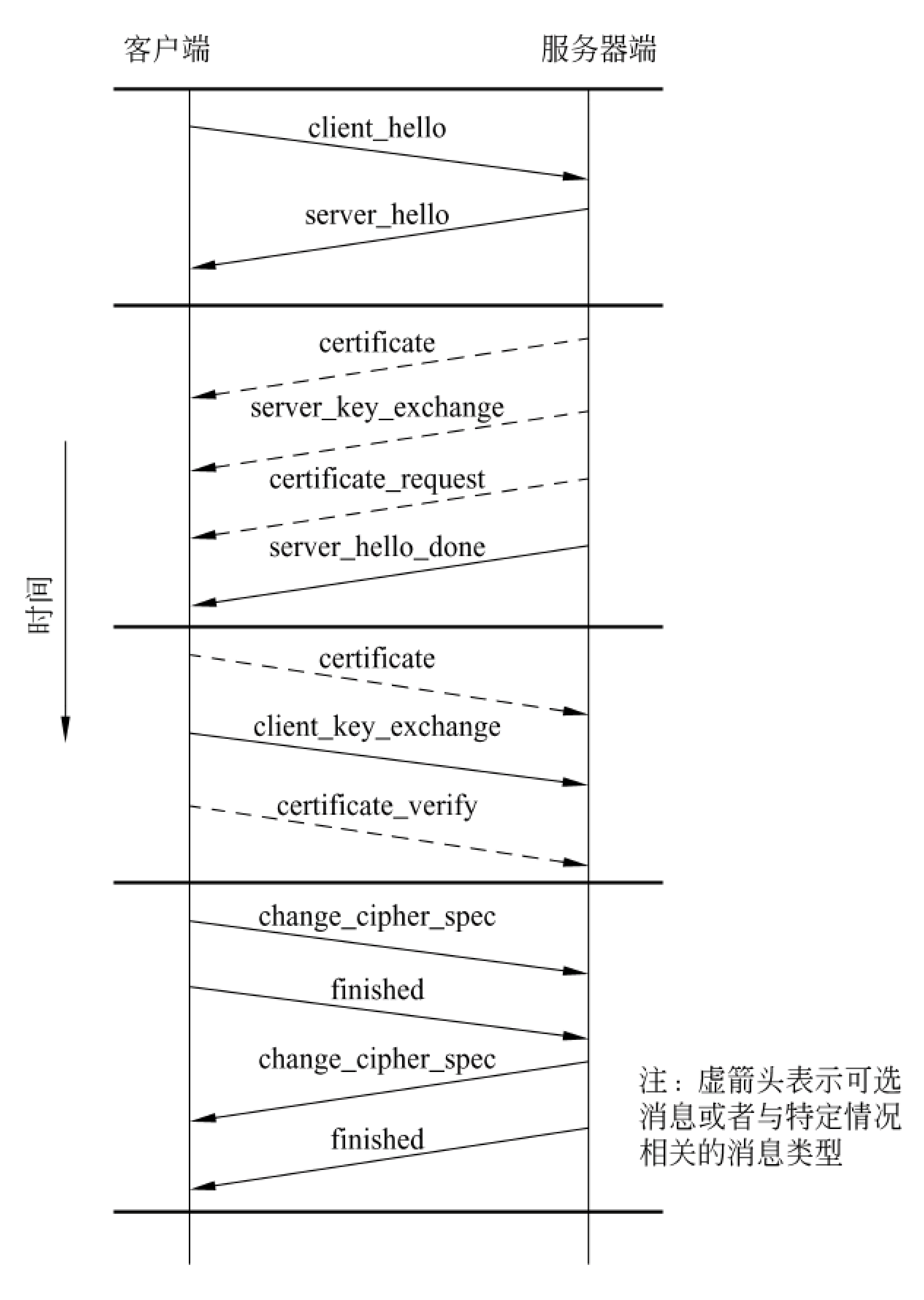


图 7-3　SSL 握手协议工作过程示意图

（1）发起阶段

客户端发起 client_hello 类型的连接请求，其中包含的参数有客户端所支持的 SSL 协议的最高版本号、随机码、会话 ID、密码套件（cipher suite，包括密钥交换算法和加密、认证算法）及压缩算法等，发出 client_hello 消息之后，客户端等待服务器的回应。服务器反馈 server_hello 类型的消息，其中版本字段是客户端和服务器都支持的最高版本号，另外还有一个与客户端相互独立的随机码，同时服务器还从客户端所提供的密码套件中确定后面所使用的密钥交换算法、加密/认证算法和压缩算法。

（2）服务器认证和密钥交换

服务器用 certificate 消息向客户端发送自己的证书，根据密钥交换算法的不同，server_key_exchange 消息包含不同算法所需参数。如果不允许匿名的客户端，则服务器发送 certificate_request 消息，要求客户端提供证书。这一阶段的最后消息是 server_done，这条消息表明服务器的相关报文已经发送完毕，接下来等待客户端的回应。

（3）客户端认证和密钥交换

收到服务器 server_done 类型的消息以后，客户端需要验证服务器提供的证书是合法

的，并且 server_hello 所包含的参数是可行的。如果服务器需要客户端提供证书的话，客户端会用 certificate 消息向服务器发送证书。在 client_key_exchange 消息中，根据协商所确定的算法，客户端向服务器发送相应的参数信息。

(4) 结束阶段

客户端和服务器分别向对方发送 change_cipher_spec 和 finished 类型的消息，至此握手的 4 个阶段全部结束，双方后续的通信全部采用协商确定的加密/认证算法和密钥进行。

3. SSL 修改密文规约协议

修改密文规约协议工作在 SSL 记录协议之上，是 SSL 协议集中最简单的，它只包含一种报文格式，报文内容只有一个字节，其值为 1，用于通知对方消息发出后发送的报文将采用握手协议中协商好的算法、密钥等进行。

4. SSL 警告协议

当通信过程中出现错误或异常情况时给出警告或者终止连接，根据错误的严重程度，Alert 消息分为两种类型：warning 和 fatal，其中如果出现 fatal 类型的消息将立即关闭连接。

7.2.2 OpenSSL 库

OpenSSL 库最早发布于 1998 年，其前身是 Eric Young 和 Tim Hudson 开发的 SSLeay 库，目前最新的版本是 0.9.8k。OpenSSL 库提供了完全的、免费的 SSL 协议实现，支持 SSL2.0、SSL3.0 以及 TSL1.0 等版本，并且能工作于大部分主流的系统平台上，如 UNIX、Linux 和 Windows 等。OpenSSL 库支持最常用的对称加密算法、公钥算法和消息摘要算法等，提供了命令行接口和编程的 API 接口。

1. Linux 环境中 OpenSSL 库的安装

从官方网站 http://www.openssl.org/上下载 OpenSSL 源代码(openssl-0.9.8k.tar.gz)。

(1) 将源代码包解压缩(代码如下)

```
#tar zxvf openssl-0.9.8k.tar.gz
```

(2) 设置安装目录并开始安装(代码如下)

```
#  cd openssl-0.9.8k
#  ./config--prefix=/usr/local/openssl
#  make
#  make install
```

2. OpenSSL 命令简介

(1) 指令 genrsa 用于生成 RSA 私有密钥，其使用格式如下：

```
openssl genrsa[-out filename][-passout arg][-des][-des3][-idea][-f4][-3][-rand file(s)][numbits]
```

指令 genrsa 选项内容的意义如下：

a. -out filename：私有密钥输入文件名，默认为标准输出。

b. -passout arg：用于输出保护 key 文件的密码。

c. -des|-des3|-idea ：加密的密钥的加密算法，一般会要输入保护密码，如果这 3 个参数中一个也没 set，密钥将不被加密。

d. -F4|-3：使用的公共组件，一种是 3，一种是 F4。

e. -rand file(s)：产生 key 的时候用过 seed 的文件，可以把多个文件用冒号分开一起做 seed。

f. numbits：指明产生的参数的长度，必须是本指令的最后一个参数。如果没有指明，则产生 512bit 长的参数。

(2) 指令 req 用来创建和处理 PKCS#10 格式的证书或者建立自签名证书，其使用格式如下：

```
openssl req[-inform PEM|DER][-outform PEM|DER][-in filename][-passin arg]
[-out filename][-passout arg][-text][-noout][-verify][-modulus][-new][-rand file(s)][-newkey rsa:bits][-
newkey dsa:file][-nodes][-key filename][-keyform PEM|DER][-keyout filename][-[md5|sha1|md2|mdc2]][-config
filename][-x509][-days n][-asn1-kludge][-newhdr][-extensions section][-reqexts section]
```

a. -inform DER|PEM：指定输入的格式是 DEM 还是 DER。DER 格式采用 ASN1 的 DER 标准格式。一般用的都是 PEM 格式，即为 base64 编码格式。PEM 格式的第一行和最后一行指明内容，中间就是经过编码的内容。

b. -outform DER|PEM：与 inform 类似，指定输出格式是 PEM 还是 DER。

c. -in filename ：要处理的 CSR 文件的名称，当-new 和-newkey 等两个 option 没有被 set，本 option 才有效。

d. -out filename：要输出的文件名。

e. -text：将 CSR 文件中的内容以可读方式打印出来。

f. -noout：不打印 CSR 文件的编码版本信息。

g. -modulus：将 CSR 中包含的公共密钥的系数打印出来。

h. -verify：检验请求文件里的签名信息。

i. -new：产生一个新的 CSR，要求用户输入创建 CSR 的一些必要的信息。需要的信息定义在 config 文件中。如果-key 没有被 set，则将根据 config 文件中的信息先产生一对新的 RSA 密钥。

j. -rand file(s)：产生 key 的时候用过 seed 的文件，可以把多个文件用冒号分开一起做 seed。

k. -newkey arg：同时生成新的私有密钥文件和 CSR 文件，本 option 是带参数的。如果是产生 RSA 的私有密钥文件，参数是一个数字，指明私有密钥 bit 的长度；如果是产生 DSA 的私有密钥文件，参数是 DSA 密钥参数文件的文件名。

l. -key filename：参数 filename 指明私有密钥的文件名，允许的格式是 PKCS#8。

m. -keyform DER|PEM：指定输入的私有密钥文件的格式是 DEM 还是 DER。DER 格式采用 ASN1 的 DER 标准格式。一般多用 PEM 格式。

n. days n：如果-x509 被设定，则该选项的参数指定 CA 给第三方签证书的有效期，默

认是 30 天。

(3) 指令 x509 是一个功能很丰富的证书处理工具。可以用来显示证书的内容,转换其格式,给 CSR 签名等,其具体格式如下:

```
openssl x509[-inform DER|PEM|NET][-outform DER|PEM|NET][-keyform DER|PEM][-CAform DER|PEM]
[-CAkeyform DER|PEM][-in filename][-out filename][-serial][-hash][-subject][-issuer]
[-nameopt option][-email][-startdate][-enddate][-purpose][-dates][-modulus]
[-fingerprint][-alias][-noout][-trustout][-clrtrust][-clrreject][-addtrust arg]
[-addreject arg][-setalias arg][-days arg][-signkey filename][-x509toreq][-req]
[-CA filename][-CAkey filename][-CAcreateserial][-CAserial filename][-text][-C][-md2|-md5|-sha1|-mdc2]
[-clrext][-extfile filename][-extensions section]
```

a. -inform DER|PEM|NET:指定输入文件的格式。

b. -outform DER|PEM|NET:指定输出文件的格式。

c. -in filename:指定输入文件名。

d. -out filename:指定输出文件名。

e. -md2|-md5|-sha1|-mdc2:指定使用的哈希算法,默认的是 MD5 与打印有关的 option。

f. -text:用文本方式详细打印该证书的所有细节。

g. -noout:不打印请求的编码版本信息。

h. -modulus:打印公共密钥的系数值

i. -serial:打印证书的序列号。

j. -hash:把证书拥有者名称的哈希值打印出来。

k. -subject:打印证书拥有者的名字。

l. -issuer:打印证书颁发者名字。

m. -nameopt option:指定用什么格式打印输出。

n. -email:如果有,打印证书申请者的 email 地址。

o. -startdate:打印证书的起始有效时间。

p. -enddate:打印证书的到期时间。

q. -dates:把以上两个 option 都打印出来。

r. -fingerprint:打印 DER 格式的证书的 DER 版本信息。

s. -C:用 C 代码风格打印结果。

t. -trustout:打印可以信任的证书。

u. -setalias arg:设置证书别名。

v. -alias:打印证书别名。

w. -clrtrust:清除证书附加项里所有有关用途允许的内容。

x. -purpose:打印证书附加项里所有有关用途允许和用途禁止的内容。

7.2.3 相关数据结构分析

BIO 是 Openssl 库中重要的数据结构。无论即将建立的 Openssl 连接安全与否,Openssl 都使用 BIO 抽象库(bio.h)来处理包括文件和套接字在内的各种类型的通信。一

个 BIO 对象就是一个 I/O 接口的抽象,它隐藏了对于一个应用的许多底层 I/O 操作的细节工作。如果一个应用使用 BIO 来进行 I/O 操作,它可以透明地处理 SSL 连接、加密连接和文件传输连接。

BIO 分为两种类型,一种是 Source/Sink 类型,一种是 Filter 类型。前者代表数据源或数据目标,例如套接字 BIO 和文件 BIO。后者的目的是把数据从一个 BIO 转换到另外一个 BIO 或应用接口,在转换过程中,这些数据可以不经过修改就进行转换。例如在加密 BIO 中,如果进行写操作,数据就会被加密;如果是读操作,数据就会被解密。

1. BIO 结构

```
typedef struct bio_st BIO;
struct bio_st
{
BIO_METHOD * method;                    //BIO方法结构,决定 BIO类型和行为的重要参数
//bio, mode, argp, argi, argl, ret
long (* callback)(struct bio_st * ,int,const char * ,int, long,long);          //BIO回调函数
char * cb_arg;                          //回调函数的第一个参数
int init;                               //初始化标识,已初始化则为 1
int shutdown;                           //开关标识,关闭为 1
int flags;                              //extra storage
int retry_reason;
int num;
void * ptr;
struct bio_st * next_bio;               //Filter 型 BIO所用,代表 BIO链的下一节
struct bio_st * prev_bio;               //Filter 型 BIO所用,代表 BIO链的上一节
int references;
unsigned long num_read;                 //读出的数据长度
unsigned long num_write;                //写入的数据长度
CRYPTO_EX_DATA ex_data;
};
```

2. 常用的 BIO 相关函数

在 BIO 的所用成员中,method 可以说是最关键的一个成员,它决定了 BIO 的类型,可以看到,在声明一个新的 BIO 结构时,总是使用下面的声明:

```
BIO * BIO_new(BIO_METHOD * type);
```

从源代码可以看出,BIO_new 函数除了给一些初始变量赋值外,主要就是把 type 中的各个变量赋值给 BIO 结构中的 method 成员。

一般来说,上述 type 参数是以一个类型生成函数的形式提供的,如生成一个 mem 型的 BIO 结构,其实现代码如下:

```
BIO * mem= BIO_new(BIO_s_mem());
```

两种 BIO 的常用相关函数如下。

(1) source/sink 型

a. BIO_s_accept():是一个封装了类似 TCP/IP socket Accept 规则的接口,并且使 TCP/IP 操作对于 BIO 接口是透明的。

b. BIO_s_bio():封装了一个 BIO 对,数据从其中一个 BIO 写入,从另外一个 BIO 读出。

c. BIO_s_connect():是一个封装了类似 TCP/IP socket Connect 规则的接口,并且使 TCP/IP 操作对于 BIO 接口是透明的。

d. BIO_s_fd():是一个封装了文件描述符的 BIO 接口,提供类似文件读写操作的功能。

e. BIO_s_file():封装了标准的文件接口的 BIO,包括标准的输入输出设备如 stdin 等。

f. BIO_s_mem():封装了内存操作的 BIO 接口,包括对内存的读写操作。

g. BIO_s_null():返回空的 sink 型 BIO 接口,写入这种接口的所有数据都被丢弃,读的时候总是返回 EOF。

h. BIO_s_socket():封装了 socket 接口的 BIO 类型。

(2) filter 型

a. BIO_f_base64():封装了 base64 编码方法的 BIO,写的时候进行编码,读的时候解码。

b. BIO_f_buffer():封装了缓冲区操作的 BIO,写入该接口的数据一般是准备传入下一个 BIO 接口的,从该接口读出的数据一般也是从另一个 BIO 传过来的。

c. BIO_f_cipher():封装了加解密方法的 BIO,写的时候加密,读的时候解密。

d. BIO_f_md():封装了信息摘要方法的 BIO,通过该接口读写的数据都是经过摘要的。

e. BIO_f_null():一个不作任何事情的 BIO,对它的操作都被简单地传到下一个 BIO 去了,相当于不存在。

f. BIO_f_ssl():封装了 openssl 的 SSL 协议的 BIO 类型,即为 SSL 协议增加了一些 BIO 操作方法。

上述各种类型的函数正是构成 BIO 强大功能的基本单元,所有这些源文件,基本上都包含于/crypto/bio/目录下扩展名为.c 的同名文件中。

3. 通过 BIO 结构进行 I/O 操作

BIO 的基本读、写操作函数主要有 4 个,它们的定义如下(openssl/bio.h):

```
int BIO_read(BIO* b, void* buf, int len);
int BIO_gets(BIO* b,char* buf, int size);
int BIO_write(BIO* b, const void* buf, int len);
int BIO_puts(BIO* b,const char* buf);
```

(1) BIO_read 函数

从 BIO 接口中读出指定数量字节 len 的数据并存储到 buf 中。成功就返回真正读出的数据的长度,失败返回 0 或-1,如果该 BIO 没有实现该函数则返回-2。

(2) BIO_gets 函数

该函数从 BIO 中读取一行长度最大为 size 的数据。通常情况下,该函数会以最大长度限制读取一行数据,但是也有例外,比如 digest 型的 BIO,该函数会计算并返回整个 digest 信息。此外,有些 BIO 可能不支持这个函数。成功就返回真正读出的数据的长度,失败返回 0 或－1,如果该 BIO 没有实现该函数则返回－2。需要注意的是,如果相应的 BIO 不支持这个函数,则对该函数的调用可能导致 BIO 链中自动增加一个 buffer 型的 BIO。

(3) BIO_write 函数

往 BIO 中写入长度为 len 的数据。成功就返回真正写入的数据的长度,失败返回 0 或－1,如果该 BIO 中没有实现该函数则返回－2。

(4) BIO_puts 函数

往 BIO 中写入一个以 NULL 为结束符的字符串,成功就返回真正写入的数据的长度,失败返回 0 或－1,如果该 BIO 中没有实现该函数则返回－2。

另外,除了这 4 个基本的 I/O 操作函数以外,还有一个比较重要的 I/O 函数。该函数也是 BIO_ctrl 的宏定义函数,其定义如下。

```
#define BIO_flush(b) (int)BIO_ctrl(b,BIO_CTRL_FLUSH,0,NULL)
```

该函数用来将 BIO 内部缓冲区的数据一次性都写出去,有些时候,也用于根据 EOF 查看是否还有数据可以写。调用成功时,该函数返回 1;失败时,返回 0 或－1。之所以失败时返回 0 或者－1,是为了标志该操作是否需要稍后以与 BIO_write()函数相同的方式重试,这时,应该调用 BIO_should_retry()函数。

需要注意的是,返回值为 0 或－1 时并不一定就是发生了错误。在非阻塞型的 source/sink 型或其他一些特定类型的 BIO 中,这仅仅代表目前没有数据可以读取,需要稍后再进行该操作。

阻塞类型的 socket 使用如 select、poll、equivalent 等函数检测 BIO 中是否存在需要被 read()函数读取的有效数据,但不建议在阻塞型的接口中使用这些技术,因为这种情况下如果调用 BIO_read()函数会导致在底层的 I/O 中多次调用 read()函数,从而导致端口阻塞。这种情况下,select(或 equivalent)应该和非阻塞型的 I/O 一起使用,可以在失败之后重新读取该 I/O,而不使端口阻塞。

7.3　实例编程练习

7.3.1　编程练习要求

在 Linux 平台上利用 OpenSSL 实现安全的 Web Server 的具体要求如下:

(1) 在理解 HTTPS 及 SSL 的工作原理的基础上,实现安全的 Web Server。

(2) Server 能够并发处理多个请求,要求至少能支持 Get 命令。可以增强 Web Server 的功能,如支持 Head、Post 以及 Delete 命令等。

(3) 编写必要的客户端测试程序,用于发送 HTTPS 请求并显示返回结果,也可以使用一般的 Web 浏览器测试。

下面给出示例程序的相关内容,供读者参考。本程序中,服务器端为命令行程序,客户端为 Linux 下普通的 Web 浏览器,这里使用的是 FireFox 浏览器。服务器端的 IP 地址为

192.168.1.91,开放的端口为22222。

程序执行的流程如下。

服务器端:./Server

```
[root@ localhost share]# ./Server
*************   Server Starting   ****************
```

此时Web服务器已经开启,等待客户端的连接请求。

在客户端,即浏览器中,键入服务器的IP地址和端口,https://192.168.1.91:22222,请求返回服务器端默认的主页。

服务器端显示从客户端发送请求到成功将所请求的文件发送回客户端之间所发生事件的日志记录(如下所示)。

```
[root@ localhost share]# ./Server
*************   Server Starting   ****************
IP:192.168.1.100 connecting to socket:872
/index.html
Closing socket:872
IP:192.168.1.100 connecting to socket:776
/index.files/winpcap.css
Closing socket:776
IP:192.168.1.100 connecting to socket:934
IP:192.168.1.100 connecting to socket:796
IP:192.168.1.100 connecting to socket:752
IP:192.168.1.100 connecting to socket:734
IP:192.168.1.100 connecting to socket:712
/index.files/New.gif
Closing socket:934
/index.files/curve.gif
Closing socket:796
/index.files/airpcap.gif
Closing socket:752
/index.files/cace_logo.gif
Closing socket:734
/index.files/curvedown.gif
Closing socket:712
```

此时,客户端已成功收到并显示所请求的网页内容(如图7-4所示)。

至此,服务器端成功完成一次基于HTTPS的Web响应。

7.3.2 编程训练设计与分析

程序流程如图7-5所示。

程序可以分为两部分:初始化模块,即通过编译及函数调用对OPENSSL库进行初始化并且创建上下文环境;Web服务模块,即基于SSL机制利用OPENSSL库函数实现

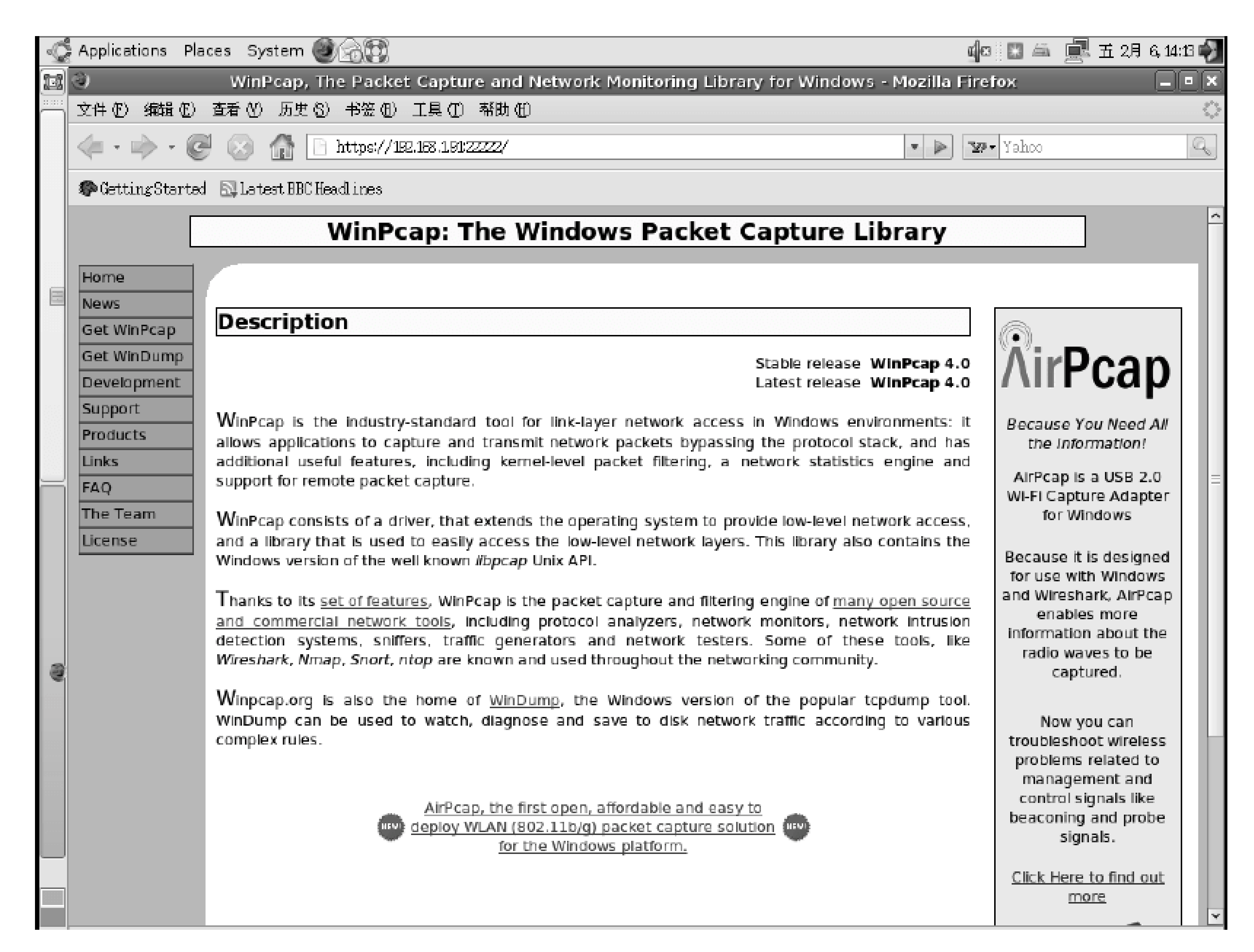


图 7-4　客户端截图

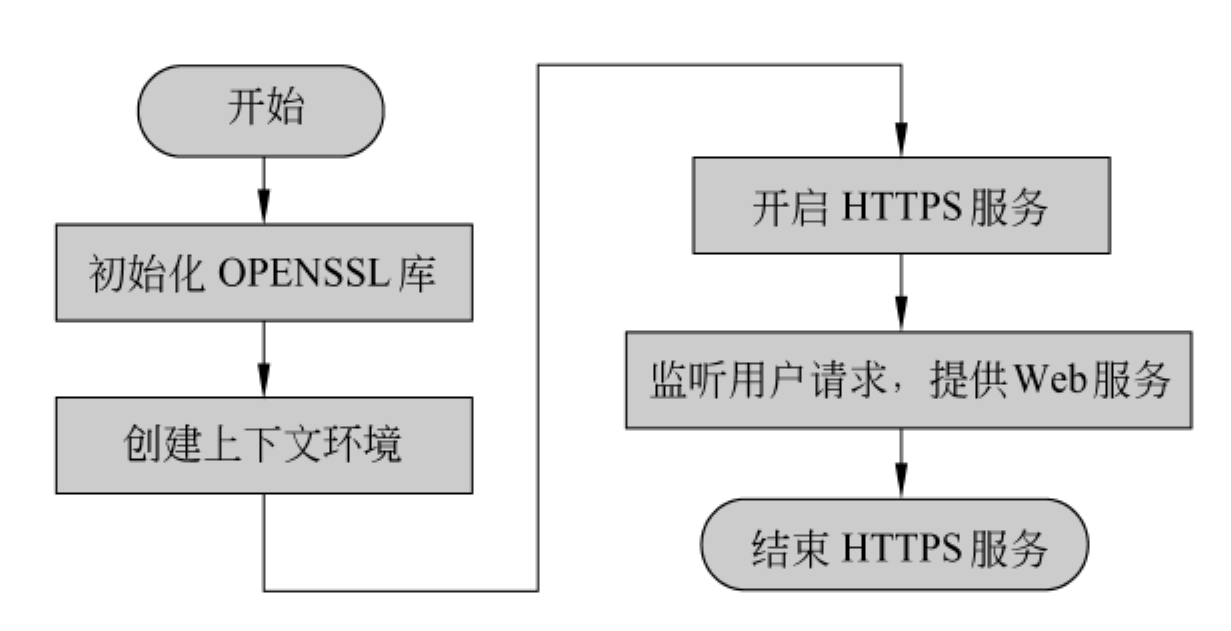


图 7-5　程序整体流程图

HTTPS 的服务。

1. 初始化模块

在开启 HTTPS 服务之前，服务器端只需要初始化 OPENSSL 库和创建上下文环境。在此过程中，会用到如下函数：

```
SSL_library_init();                  //加载 OPENSSL将会用到的算法
SSL_load_error_strings();            //加载错误字符串
SSL_METHOD* meth;
SSL_CTX* ctx;                        //SSL_CTX 对象
meth= SSLv23_method();               //相应的 SSL结构使用的是 SL2.0、3.0,但可以回到 SSL2.0
ctx= SSL_CTX_new(meth);              //创建一个上下文环境
SSL_CTX_use_certificate_chain_file(ctx, "server.pem");    //指定所使用的证书文件
SSL_CTX_set_default_passwd_cb(ctx, password_cb);          //设置密码回调函数
```

```
SSL_CTX_use_PrivateKey_file(ctx, "server.pem", SSL_FILETYPE_PEM);           //加载私钥文件
SSL_CTX_load_verify_locations(ctx, "root.pem", 0);          //加载受信任的 CA 证书
load_dh_params(ctx, "dh1024.pem");
//当使用 RSA 算法鉴别的时候,会有一个临时的 DH 密钥磋商发生。这样会话数据将用
//这个临时的密钥加密,而证书中的密钥作为签名
```

2. Web 服务模块

Web 服务模块是本程序的核心部分,其执行流程如图 7-6 所示。

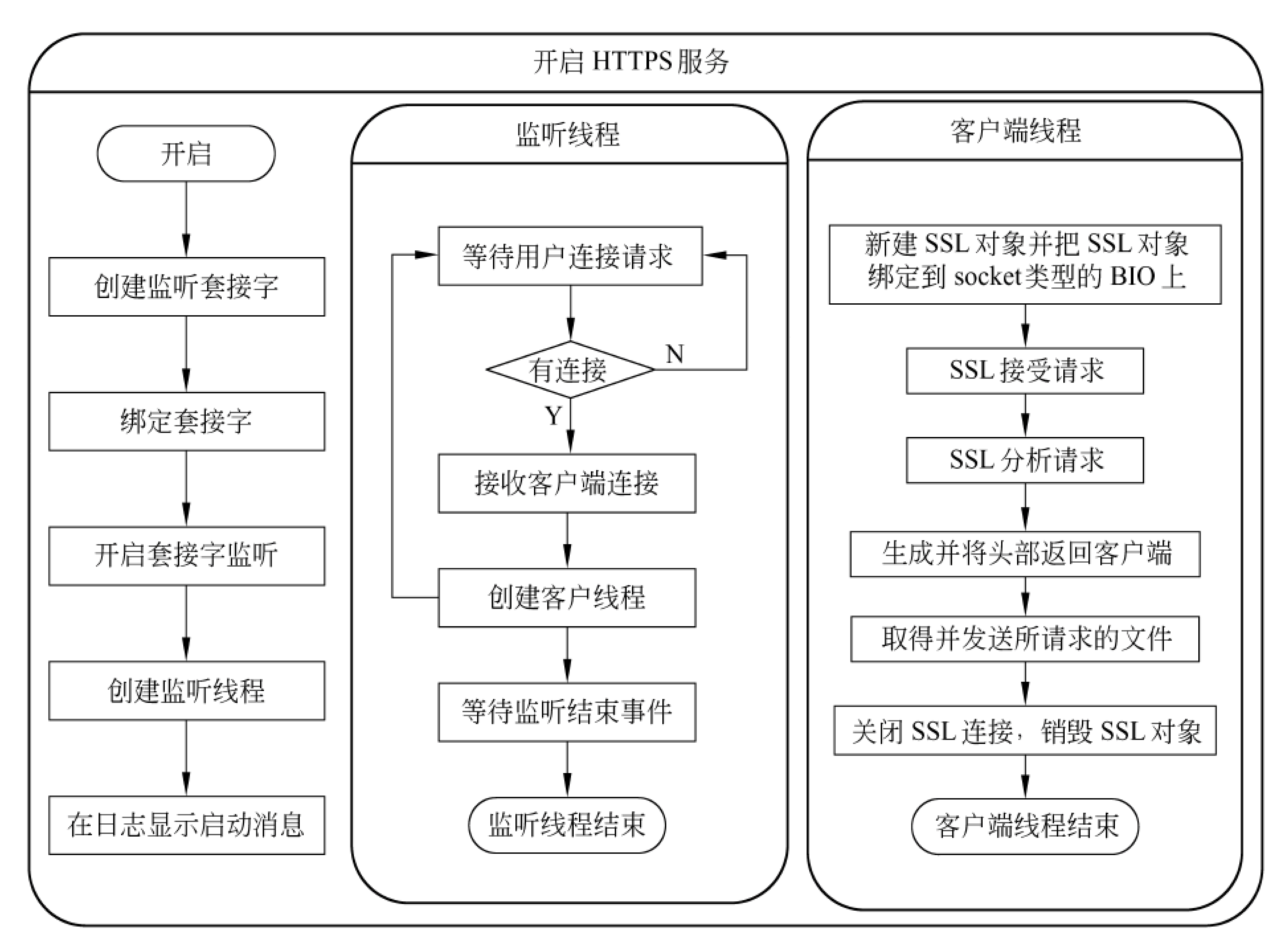


图 7-6　HTTPS 实现流程图

Web 服务模块在类 CHttpProtocol 中实现。该类中封装了 HTTPS 中与本程序相关的操作,主要包括监听线程函数以及客户端线程函数中需要调用的子函数的定义及实现。其中,比较核心的子函数包括 SSL 分析请求,将响应头部返回给客户端以及将文件发送回客户端等。

下面给出该类定义中的部分代码:

```
class CHttpProtocol
{
…
    CHttpProtocol(void);
    ~CHttpProtocol(void);
    SSL_CTX* ctx;                                              //SSL上下文
    char* initialize_ctx();                                    //初始化 CTX
    char* load_dh_params(SSL_CTX* ctx, char* file);            //加载 CTX参数
```

```
    int TcpListen();                                   //TCP 监听函数
    void StopHttpSrv();                                //停止 HTTP 服务
    bool StartHttpSrv();                               //开始 HTTP 服务
    static void* ListenThread(LPVOID param);           //监听线程
    static void* ClientThread(LPVOID param);           //客户线程
//接收 HTTPS 请求
    bool SSLRecvRequest(SSL* ssl,BIO* io, LPBYTE pBuf, DWORD dwBufSize);
int Analyze(PREQUEST pReq, LPBYTE pBuf);               //分析 HTTP 请求
    bool SSLSendHeader(PREQUEST pReq, BIO* io);        //发送 HTTPS 头
    bool SSLSendFile(PREQUEST pReq, BIO* io);          //由 SSL 通道发送文件
    bool SSLSendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize);
…
};
```

下面对本示例程序中的模块依次进行介绍。

(1) 监听线程

监听线程用于监听 Web 服务连接请求。

使用 pthread_create()函数创建监听线程,代码如下:

```
pthread_t listen_tid;
pthread_create(&listen_tid,NULL,&ListenThread,this);
```

其中传递给监听线程函数的参数是调用该成员函数的类对象指针,监听函数可以由这个类对象指针调用其需要的信息,如套接字句柄、SSL 上下文等。

在监听线程函数中,服务器端循环等待客户端的连接请求,若有新的请求到达,服务器端将新创建一个客户端线程去处理该客户端的请求。

```
void* CHttpProtocol::ListenThread(LPVOID param)
{…
    CHttpProtocol * pHttpProtocol= (CHttpProtocol * )param;
    while(1)            //循环等待,如有客户连接请求,则接受客户机连接要求
    {
        nLen= sizeof(SockAddr);
        //套接字等待连接,返回对应已接受的客户机连接的套接字
        socketClient= accept(pHttpProtocol->m_listenSocket, (LPSOCKADDR)&SockAddr, &nLen);
        if (socketClient== INVALID_SOCKET)
            break;
            //创建 client 进程,处理 request
        pthread_create(&client_tid,NULL,&ClientThread,pReq);
    }
…
}
```

(2) 客户端线程

客户端线程用于处理接收到的客户端请求。

客户端线程由监听线程在 accept()函数调用成功后创建。在客户端线程函数中,首先

将根据之前初始化的上下文创建出来的 SSL 对象绑定在一个 socket 类型的 BIO 上面，然后调用 SSL_accept()函数，与客户端进行握手。

下面的代码摘自客户端线程函数：

```
void* CHttpProtocol::ClientThread(LPVOID param)
{
…
    PREQUEST pReq= (PREQUEST)param;
    CHttpProtocol * pHttpProtocol= (CHttpProtocol * )pReq->pHttpProtocol;
    SOCKET s=pReq->Socket;
    sbio=BIO_new_socket(s, BIO_NOCLOSE);          //创建一个 socket 类型的 BIO 对象
    ssl=SSL_new(pReq->ssl_ctx);                   //创建一个 SSL 对象
    SSL_set_bio(ssl, sbio, sbio);                 //把 SSL 对象绑定到 socket 类型的 BIO 上
    //连接客户端，在 SSL_accept 过程中，将会占用很大的 cpu
    nRet=SSL_accept(ssl);
    //nRet<=0 时发生错误
    io=BIO_new(BIO_f_buffer());                   //封装了缓冲区操作的 BIO
    ssl_bio=BIO_new(BIO_f_ssl());
    //封装了 SSL 协议的 BIO 类型，即为 SSL 协议增加了一些 BIO 操作方法
    BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE);
    //把 ssl(SSL 对象)封装在 ssl_bio(SSL_BIO 对象)中
    BIO_push(io, ssl_bio);
    //把 ssl_bio 封装在一个缓冲的 BIO 对象中，实现对 SSL 连接的缓冲的读和写
    …
    //做好上述 I/O 绑定之后，开始接受客户端请求
    if (!pHttpProtocol->SSLRecvRequest(ssl,io,buf,sizeof(buf)))
    {
        //处理错误
        …
    }
    //HTTPS 协议分析
    nRet=pHttpProtocol->Analyze(pReq, buf);
    if (nRet)
    {
        //处理错误
        …
    }
    //生成并返回头部
    if(!pHttpProtocol->SSLSendHeader(pReq,io))
    {
        //处理错误
        …
    }
    BIO_flush(io);
    //向 client 传送数据
```

```
    if(pReq->nMethod==METHOD_GET)
    {
        if(!pHttpProtocol->SSLSendFile(pReq,io))
        {
            //处理错误
            …
        }
    }
    //析构操作
…
}
```

客户端线程的参数定义如下：

```
typedef struct REQUEST
{
    SOCKET          Socket;                     //请求的 socket
    int             nMethod;                    //请求的使用方法：GET 或 HEAD
    DWORD           dwRecv;                     //收到的字节数
    DWORD           dwSend;                     //发送的字节数
    int             hFile;                      //请求连接的文件
    char            szFileName[256];            //文件的相对路径
    char            postfix[10];                //存储扩展名
    SSL_CTX* ssl_ctx;                           //SSL 上下文
    void* pHttpProtocol;                        //指向类 CHttpProtocol 的指针
    …
    }REQUEST, * PREQUEST;
```

(3) 接受客户端请求

接受客户端请求的工作由函数 SSLRecvRequest()完成，部分代码如下：

```
bool CHttpProtocol::SSLRecvRequest(SSL* ssl,BIO* io, LPBYTE pBuf, DWORD dwBufSize)
{   …
    memset(buf, 0, BUFSIZZ);                    //初始化缓冲区
    while(1)
    {
        r=BIO_gets(io, buf, BUFSIZZ-1);
        switch(SSL_get_error(ssl, r))
        {
            case SSL_ERROR_NONE:
                memcpy(&pBuf[length], buf, r);
                length+=r;
                break;
            default:
                break;
        }
        //直到读到代表 HTTP 头部结束的空行
```

```
            if(!strcmp(buf,"\r\n")||!strcmp(buf,"\n"))
                break;
        }
        //添加结束符
        pBuf[length]='\0';
        return true;
    }
```

该函数首先利用 BIO_gets()函数从 I/O 中读取数据，放到事先分配好的缓冲区 buf 里，并通过对换行符的判断以确保获得完整的 HTTPS 头部，以便对缓冲区中的 HTTPS 请求数据报进行分析。

(4) HTTPS 协议解析

下面代码片段摘自 SSL 请求分析函数，由客户端线程函数调用：

```
int  CHttpProtocol::Analyze(PREQUEST pReq, LPBYTE pBuf)
{
    //分析接收到的信息
    char szSeps[]="\n";
    char * cpToken;
    //判断 request 的 method
    cpToken=strtok((char * )pBuf, szSeps);          //缓存中字符串分解为一组标记串
    if (!strcmp(cpToken, "GET"))                    //GET 命令
    {
        pReq->nMethod=METHOD_GET;
    }
    …
    strcpy(pReq->szFileName, m_strRootDir);
    if (strlen(cpToken)>1)
    {
        strcat(pReq->szFileName, cpToken);          //把该文件名添加到结尾处形成路径
    }
    else
    {
        strcat(pReq->szFileName, "/index.html");    //若无文件名,则默认为 index.html
    }
    …
}
```

该函数根据 HTTPS 协议的标准请求格式的头部来进行分解与分析，获取请求的命令以及请求的文件等信息。服务器根据这些信息，将相应的响应及文件本身发送回客户端。

(5) 发送 HTTPS 响应

服务器端会首先根据客户端请求的命令和文件，来判断命令是否符合标准，所请求的文件是否存在，然后组成一个响应发回客户端。人们通常在上网的时候，网页上出现 404 或 400 等错误，就是因为这一步请求的文件于服务器上不存在或者命令错误，服务器就会发送一个错误的响应到用户的浏览器上。

下面的代码摘自服务器端的响应函数，该函数由客户端线程函数调用。

```
bool CHttpProtocol::SSLSendHeader(PREQUEST pReq, BIO* io)
{
    …
    char curTime[50];
    GetCurrentTime(curTime);
    //取得文件的 last-modified 时间
    char last_modified[60]=" ";
    GetLastModified(pReq->hFile, (char* )last_modified);
    //取得文件的类型
    char ContentType[50]=" ";
    GetContentType(pReq, (char* )ContentType);
    //组成完整的服务器响应
    sprintf((char* )Header, "HTTP/1.1 %s\r\nDate: %s\r\nServer: %s\r\nContent-Type: %s\r\nContent-
  Length: %d\r\n\r\n",
        HTTP_STATUS_OK,
        curTime,                              //Date
        "Villa Server 192.168.1.91",          //Server"My Https Server"
        ContentType,                          //Content-Type
        length);                              //Content-length
    …
    BIO_flush(io);                            //一次性清空缓冲区,全部写入 I/O
    return true;
}
```

(6) 释放相关资源

在客户端线程完成请求之后，需要释放相关资源，代码如下：

```
SSL_shutdown(ssl);              //关闭 SSL 连接
SSL_free(ssl);                  //释放 SSL 结构
SSL_CTX_free(ssl_ctx);          //释放上下文环境
```

7.4　扩展与提高

7.4.1　客户端认证

建立 SSL 连接时，一般要求服务器提供认证证书，由客户端验证通过才能继续建立连接，客户端一般并不被要求提供客户端的认证证书，本 Web 服务器没有要求客户端提供认证证书。作为扩展，可以为服务器添加客户端认证功能，即进行客户端和服务器的双向认证。

采用双向认证方式的 SSL 连接，既要求服务器提供认证证书，由客户端验证，同时也要求客户端提供自己的认证证书，由服务器进行验证。

OpenSSL 提供了建立双向认证 SSL 连接的 API，只需要在单向认证的基础上，设置要求对方提供证书的属性，并设置系统可信的证书，OpenSSL 提供的相关 API 如下。

(1) 设置可信 CA 的证书文件(如下所示)

```
int SSL_CTX_load_verify_locations(SSL_CTX* ctx, const char* cafile, const char* cadir);
```

(2) 设置认证模式(如下所示)

```
int SSL_CTX_set_verify(SSL_CTX* ctx, int mode,
                       int (* verify_callback)(int, X509_STORE_CTX* ));
```

其中 mode 参数表示认证模式,verify_callback 提供注册认证回调函数的机制,为 NULL 时使用 OpenSSL 内置的认证函数。使用双向认证模式需要将 mode 的值置为 SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT。

(3) 设置可信证书链深度(如下所示)

```
void SSL_CTX_set_verify_depth(SSL_CTX* ctx, int depth);
```

如果需要验证客户端,则在服务器中添加如下代码即可。

```
SSL_CTX_load_verify_locations(ctx, RSA_SERVER_CA_CERT/* 客户证书的根 CA* /, NULL);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
SSL_CTX_set_verify_depth(ctx, 1);
```

7.4.2 基于 IPSec 的安全通信

利用 SSL 可以保证 Web 浏览器和 Web 服务器间的安全通信,利用 PGP(Pretty Good Privacy)及 S/MIME(Secure/Multipurpose Internet Mail Extension)可以实现邮件加密,但是这些安全技术都只能用于局部业务,并不能保证 TCP/IP 整体上的安全通信,因此出现了能够使企业和个人用户在开放的 Internet 上通用的安全协议——IPSec。

IPSec 协议是 IETF 于 1998 年 11 月公布的 IP 安全标准,目前 IPSec 被广泛应用于实现端到端的安全、虚拟专用网和安全隧道,它对 IPv4 是可选的,对于 IPv6 则是强制必须实施的,是唯一一种可为任何形式的 Internet 通信提供安全保障的协议,也是易于扩展的、完整的一种基础网络安全方案。IPSec 协议是一个协议族,它包括验证头协议(Authentication Header,AH)、封装安全载荷协议(Encapsulation Security Payload,ESP)和 Internet 密钥交换协议(Internet Key Exchange,IKE)等。

1. IPSec 体系结构

IPSec 提供了一种标准、健壮且包容广泛的安全机制,可以为 IP 及其上层协议提供安全保证。其具体保护形式有:数据源验证、无连接数据的完整性验证、数据机密性、抗重播及有限的数据流机密性保证。IPSec 是一个协议族,其结构如图 7-7 所示。

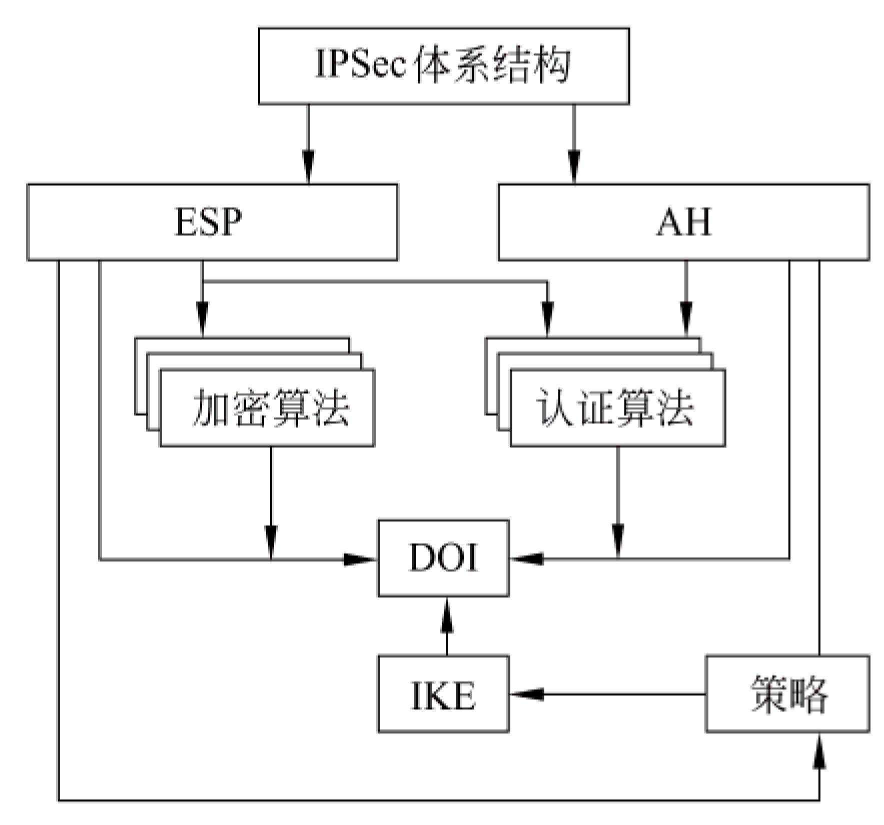


图 7-7 IPSec 安全体系结构示意图

ESP 和 AH:ESP 是插入 IP 数据包内的一个协议头,可以为 IP 提供机密性、数据源验证、抗重播以及数据完整性等安全服务。AH 与

ESP 类似，但它不提供机密性服务。

策略：目前尚未形成标准，它决定两个实体之间是否能够通信以及如何通信，IPSec 策略由安全策略数据库(Security Policy Database，SPD)维护。一个 SPD 条目可能定义了下述几种行为之一：丢弃、绕过及应用。对那些定义了应用行为的 SPD 条目，它们均会指向一个或一串安全关联(Security Association，SA)。SA 定义了对一个特定的 IP 包作 IPSec 处理对应的各种安全参数，包括要应用的安全协议(ESP、AH)、算法及密钥、密钥生存期、抗重播窗口等。

IKE：SA 可以手工或动态建立，IKE 用于动态建立 SA，它可以代表 IPSec 对 SA 进行协商。

(1) ESP

ESP 是属于 IPSec 的一种协议，协议号为 50，用于保护 IP 数据包的机密性、数据的完整性以及数据源的身份认证，也负责抵抗重播攻击。ESP 用一个加密器来提供机密性，一个身份验证器来保证完整性。对于发送的数据包，首先进行加密处理，然后是验证处理；对于接收的数据包，操作相反。应用 ESP 时，在 IP 报头之后，要保护的数据之前插入一个 ESP 头，之后插入一个 ESP 尾。ESP 保护的数据包的封装格式如图 7-8 所示。

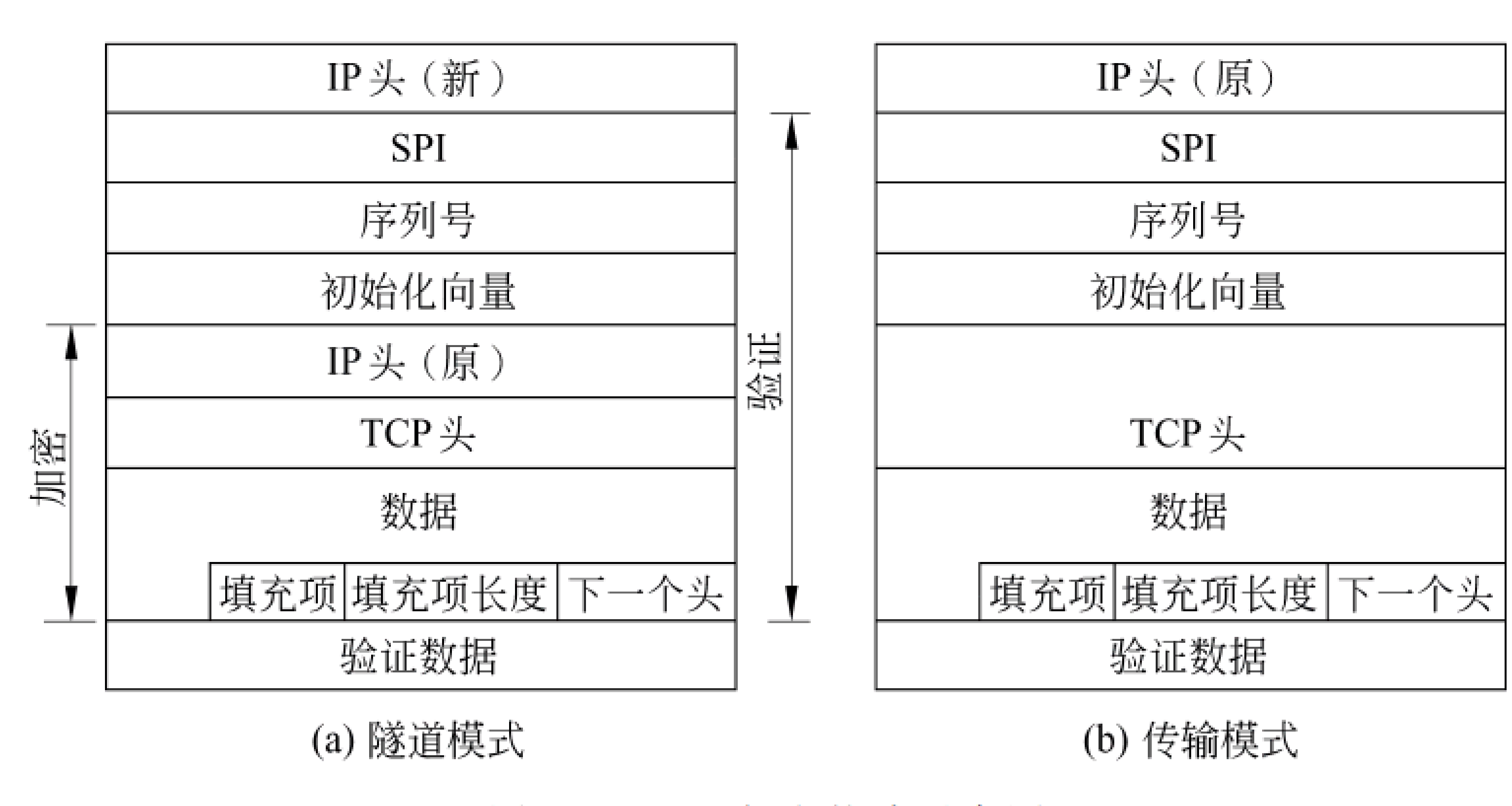


图 7-8 ESP 报文格式示意图

a. 安全参数索引(Security Parameter Index，SPI)：用来和对端的安全设备同步使用加密算法和认证算法。通常接收端使用元组＜SPI、目标主机、协议＞唯一标识所使用的 SA。

b. 序列号：单向递增的计数器，用来抗重播攻击。

c. 初始化向量(Initialization Vector，IV)：这是一个可选的 32 位字段，会出现 3 种不同的情况：加密算法不需要 IV、需要隐式的 IV 和需要显式的 IV。

d. 填充项：0～255 字节，用于保证边界的正确性。某些加密算法模式对加密数据长度有要求；即使没有要求机密性保证，仍需要把“下一个头”字段靠右排列；同时，这项技术隐藏了原始数据的实际长度。

e. 填充项长度：给出前面的填充项的长度，置 0 时表示没有填充。

f. 下一个头：表明数据类型。在隧道模式下该值为 4，表示 IP-in-IP；在传输模式下，该值表示载荷所属协议的类型，例如 TCP 协议对应的值为 6。

g. 验证数据：用于容纳数据完整性的验证结果，即完整性验证值(Integrity Check

Value,ICV)。

ESP 的工作机制是：发送方先对数据包进行加密，再用信息摘要算法计算验证数据，并作为该 IP 分组的一部分一起转发出去。在分组的接收端，重新计算该摘要并与原摘要进行比较，验证成功后，进行解密。

(2) AH

AH 也是一种 IPSec 协议，协议号为 51，用于为 IP 提供数据完整性，数据原始身份验证和一些可选的、有限的抗重播服务。AH 不提供任何保密性服务，它不加密所保护的数据包，因此 AH 头比 ESP 头简单得多。由于不需要填充，不需要填充长度指示器，因此也不存在尾，另外，也不需要初始化向量。图 7-9 描述了隧道模式下受 AH 保护的 IP 报文格式。

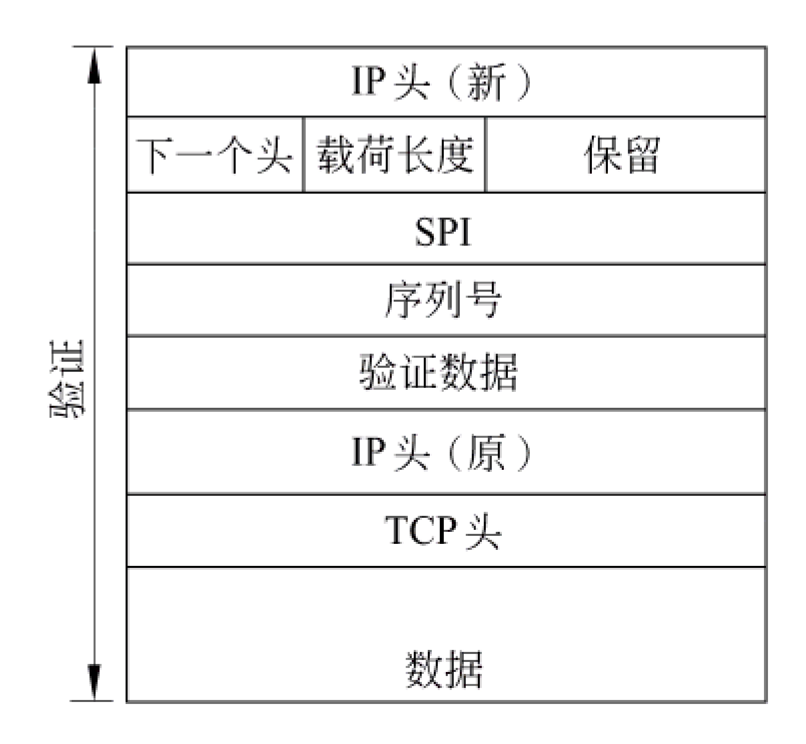


图 7-9 隧道模式下的 AH 报文格式示意图

a. 载荷长度：表示以 32 比特为单位的验证头的长度减去 2，并不单指 IP 包的实际负载长度。

b. 下一个头、SPI 和序列号：这几个字段与 ESP 中的意义相同。

c. 验证数据：用于容纳数据完整性的验证结果(ICV)。计算出 ICV 之前，该字段置为 0。与 ESP 不同，AH 将完全保护扩展到外部 IP 头的恒有或预计有的字段，因此，在做 ICV 计算时，可变字段必须全部置为 0。可变字段包括 IP 头中服务类型、分段偏移、TTL、头校验和等。

AH 的工作机制是：利用单向的信息摘要算法，对整个 IP 分组或上层协议计算一个摘要并作为该 IP 分组的一部分一起转发出去，在分组的接收端，重新计算该摘要并与原摘要进行比较，如果分组在传输过程中被修改过，则会由于摘要不一致而被丢弃，从而实现数据源鉴别和数据的完整性保护。

(3) 安全策略和安全关联

a. 安全策略(Security Policy,SP)是 IPSec 中尚未成为标准的一个重要组件，它决定了为一个数据包提供的安全服务，存放在安全策略数据库 SPD 中。IP 包的发送和接收处理都要以安全策略为准，并且，要求策略管理应用能够添加、删除和修改策略，但 SPD 的具体管理方式由实现方案决定，并未为此专门定义一套统一的标准。

每个 SPD 中的条目由选择符和策略项组成，根据选择符(selector)对 SPD 进行检索，得到相应的策略。选择符是从网络层和传输层头内提取出来的，包括源地址、目的地址、传输协议、上层端口等。策略项一般是 3 种不同的行为之一：丢弃、绕过和应用 IPSec。当采取应用 IPSec 这一行为时，策略会提供一个三元组或四元组，可以称作 SAID：＜目的地址(外部地址)、安全协议(ESP 或 AH)、SPI＞或＜源地址、目的地址、安全协议、SPI＞。这个 SAID 作为检索 SA 以获得详细安全参数的依据。

b. SA 是构成 IPSec 的基础。SA 是两个通信实体经协商建立起来的一种协定。它们决定了用来保护数据包安全的 IPSec 协议(ESP 或 AH)、转码方式、密钥以及密钥的有效存在时间等。任何 IPSec 实施方案始终会构建一个安全关联数据库(Security Association Database,SAD)，由它来维护 IPSec 协议用来保障数据包安全的 SA 记录。

SA 是单向的。如果两个主机(比如 A 和 B)正在通过 ESP 进行安全通信,那么主机 A 就需要有一个 SA,即 SA (out),用来处理往外发的数据包;另外还需要有一个相对应的 SA,即 SA (in),用来处理进入的数据包。主机 A 的 SA (out)和主机 B 的 SA (in)将共享相同的安全参数(比如密钥)。类似地,主机 A 的 SA (in)和主机 B 的 SA (out)也会共享同样的安全参数。

每个 SAD 中的条目也可以看成由两部分组成:SAID 和 SA 记录。SA 记录中主要包括序号计数器、算法、密钥、SA 的 TTL、IPSec 模式等。

c. 在包的处理过程中,SPD 和 SAD 这两个数据库需要联合使用。外出处理时先检索 SPD,进入处理时先检索 SAD。

对一个发送的数据包而言,根据提取的选择符检索 SPD,它会命中某个 SP。根据策略,包可能被丢弃、直接传送或应用 IPSec 安全服务;当需要 IPSec 安全服务时,SP 返回一个 SAID,用 SAID 检索 SAD,最终命中 SA。根据 SA 指明的各项安全参数对数据包进行安全封装。接收处理有别于发送处理,以下两种情况应分别对待:①如果收到的数据包内没有包含 IPSec 头,说明是个普通 IP 包:检索 SPD,根据检索结果决定将包丢弃,或者传递给传输层做进一步的处理。②IP 包中含有 IPSec 头:提取 IPSec 头部信息,组成 SAID,检索 SAD,根据 SA 中指明的各项安全参数对数据包进行验证、解封装。接着,根据从解封了的原始 IP 包中提取的选择符检索 SPD,验证 SA 的使用是否得当。

(4) IKE

IPSec 默认的自动密钥管理协议是 IKE。IKE 是 Oakley 和 SKEME 协议的一种混合,并在由 ISAKMP 规定的框架内运作。

IKE 用于动态建立 SA。IKE 代表 IPSec 对 SA 进行协商,最终确定可以称为“保护套件”的安全参数——包括加密算法、散列算法、验证方法和 Diffie-Hellman 组。之后对安全关联库 SAD 进行填充。IKE 是一个用户级进程,启动后作为后台守护进程运行,在需要使用 IKE 服务前它一直处于不活动状态。

可以通过以下两种方式来请求 IKE 服务:

a. 内核的安全策略模块要求自动建立 SA 时;

b. 远程 IKE 实体需要协商 SA 时。

IKE 使用了两个阶段的 ISAKMP。在第一阶段,通信各方彼此间建立一个已经通过身份验证和安全保护的通道,即建立 IKE 安全联盟。在第二阶段,利用这个既定的安全联盟,为 IPSec 协商具体的安全联盟。

IKE 共定义了 5 种交换:两个阶段 1 次交换,1 个阶段两次交换,两个额外的交换。阶段一的两种交换:对身份进行保护的“主模式”交换,根据基本 ISAKMP 文档制订的“野蛮模式”交换。阶段二使用“快速模式”交换。IKE 自己定义了两种交换:①为通信各方之间协商一个新的 Diffie-Hellman 组类型的“新组模式”交换;②在 IKE 通信双方间传送错误及状态消息的 ISAKMP 信息交换。

IKE 保证了动态建立安全联盟和建立过程的安全。IKE 的实现是 IPSec 协议实现的重要组成部分,其实现极为复杂;但它也很可能成为整个系统的瓶颈。优化 IKE 程序、优化密钥算法是实现 IPSec 的核心问题之一。

2. IPSec 工作模式

IPSec 协议既可以用来保护 IP 上层协议，也可以用来保护一个完整的 IP 数据包。这两方面的保护分别由 IPSec 的传输模式和隧道模式来提供。数据包格式如图 7-10 所示。

原始数据包：

IP 头	TCP 头	数据

传输模式下的数据包：

IP 头	IPSec 头	TCP 头	数据	IPSec 尾

隧道模式下的数据包：

新 IP 头	IPSec 头	IP 头	TCP 头	数据	IPSec 尾

图 7-10　两种 IPSec 模式的数据包结构示意图

只有在要求端到端的安全保障时，才能使用 IPSec 的传输模式。路由器主要通过检查网络层来做出路由决定，而且路由器不会、也不应该改变网络层头之外的其他东西。如果通过路由器为数据包流插入传输模式的 IPSec 头，便违反了这一规则。

在隧道模式中，把要保护的整个 IP 包封装到另一个 IP 数据包里，增加了一个新的 IP 头。通信终点由内部 IP 头指定，而加密终点，即隧道终点，则由外部新的 IP 头指定。

第 8 章 网络端口扫描器的设计与编程

8.1 本章训练目的与要求

网络端口扫描器是一种重要的网络安全检测设备，也是黑客进行网络攻击的重要工具之一。通过端口扫描，不仅可以发现目标主机的开放端口和操作系统的类型，还可以查找系统的安全漏洞，获得口令缺陷等相关信息。因此，掌握端口扫描基本工作原理与软件设计方法是信息安全工程师必须掌握的基本技能之一。同时，研究网络端口扫描器的实现方法，对于维护网络系统的安全，了解黑客攻击的手段有着重要的意义。

本章训练的主要目的是：

(1) 理解网络端口扫描器的基本结构、工作原理与设计方法。

(2) 掌握 TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描，以及 UDP 扫描的基本工作原理、设计与实现方法。

(3) 掌握 ping 程序的设计与实现方法。

(4) 掌握 Linux 操作系统多线程编程的基本方法。

本章编程训练的要求如下：

(1) 编写端口扫描程序，实现 TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描等 4 种基本的扫描方式。

(2) 设计并实现 ping 程序，探测目标主机是否可达。

8.2 相关背景知识

8.2.1 ping 程序

ping 程序是日常网络管理中经常使用的程序。它用于确定本地主机与网络中其他主机的通信情况。因为只是简单地探测某一 IP 地址所对应的主机是否存在，因此它的原理十分简单。扫描发起主机向目标主机发送一个要求回显(type=8，即为 ICMP_ECHO)的 ICMP 数据包，目标主机在收到请求后，会返回一个回显(type=0，即为 ICMP_ECHOREPLY)的 ICMP 数据包。扫描发起主机可以通过是否接收到响应的 ICMP 数据包来判断目标主机是否存在。

在本章编程中，可以在向目标主机发起端口扫描之前使用 ping 程序确定目标主机是否存在。如果 ping 目标主机成功，则继续后面的扫描工作；否则，放弃对目标主机的扫描。

8.2.2 TCP 扫描

1. TCP 连接建立过程

图 3-3 已经给出了 TCP 报头的结构。对于 TCP 端口扫描而言,TCP 协议的标志位起着至关重要的作用。虽然 TCP 扫描的种类繁多,但是它们的原理都与 TCP 连接的建立过程密切相关。在介绍各种 TCP 扫描之前,必须深入理解一个 TCP 连接的建立过程。该过程的步骤如下:

(1) 首先客户端(请求方)在连接请求中,向服务器端(接收方)发送 SYN=1,ACK=0 的 TCP 数据包,表示希望与服务器建立一个连接。

(2) 如果服务器端响应这个连接请求,就返回一个 SYN=1,ACK=1 的数据包给客户端,表示服务器端同意建立这个连接,并要求客户端进行确认。

(3) 最后客户端发送一个 SYN=0,ACK=1 的数据包给服务器端,表示确认建立连接。

2. 与 TCP 协议相关的 3 种扫描

(1) connect 扫描

TCP connect 扫描非常简单。扫描发起主机只需要调用系统 API connect 尝试连接目标主机的指定端口,如果 connect 成功,那么意味着扫描发起主机与目标主机之间至少经历了一次完整的 TCP 三次握手建立连接过程,因此被测端口是开放的;否则表示被测端口关闭。

虽然在编程时不需要程序员手动构造 TCP 数据包,但是 connect 扫描的效率非常低。由于 TCP 协议是可靠传输协议,connect 系统调用不会在尝试发送第一个 SYN 包未得到响应的情况下就放弃,而是会经过多次尝试后才彻底放弃,因此需要较长的时间。此外,connect 失败会在系统中造成大量的连接失败日志,容易被管理员发现。

(2) SYN 扫描

TCP SYN 扫描是使用最广泛的扫描方式,其原理就是向待扫描端口发送 SYN 数据包。如果扫描发起主机能够收到 ACK|SYN 数据包,则表示端口开放;如果收到 RST 数据包,则表示端口关闭。如果未收到任何数据包,且确定目标主机存在,则发送给被测端口的 SYN 数据包可能被防火墙等安全设备过滤。因为 SYN 扫描不需要完成 TCP 连接的三次握手过程,所以它又被称为半开放扫描。

SYN 扫描的最大优点就是速度快。在 Internet 中,如果不考虑防火墙的影响,SYN 扫描每秒钟可以扫描数千个端口。但是由于其扫描行为较为明显,SYN 扫描容易被入侵检测系统发现,也容易被防火墙屏蔽。同时构造原始的 TCP 数据包也需要较高的系统权限(在 Linux 中仅限于 root 账户)。

(3) FIN 扫描

TCP FIN 扫描会向目标主机的被测端口发送一个 FIN 数据包。如果目标主机没有任何响应且确定该主机存在,则表示目标主机正在监听这个端口,端口是开放的;如果目标主机返回一个 RST 数据包且确定该主机存在,则表示目标主机没有监听这个端口,端口是关闭的。

FIN 扫描具有良好的隐蔽性,不会留下日志。但是它的应用具有很大的局限性。由于不

同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 Windows 系统而言，由于无论其端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断。

8.2.3　UDP 扫描

一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可达(ICMP port unreachable)的错误。UDP 扫描就是利用了上述原理，向被扫描端口发送 0 字节的 UDP 数据包，如果收到一个 ICMP 不可达响应，则认为端口是关闭的；而对于那些长时间没有响应的端口，则认为是开放的。

但是，因为大部分系统都限制了 ICMP 差错报文的产生速度，所以针对特定主机的大范围 UDP 端口扫描的速度非常缓慢。此外，UDP 协议和 ICMP 协议是不可靠协议，数据包丢失也可能造成没有收到响应的情况，因此扫描程序必须对同一端口进行多次尝试后才能得出正确的结论。

8.2.4　使用原始套接字构造并发送数据包

本书第 6 章已经对原始套接字(SOCK_RAW)进行了简单介绍。与流套接字和数据报套接字相比，原始套接字的独特之处在于程序员可以创建并填充协议头。在编写端口扫描器程序时，除了 TCP connect 扫描可以直接调用 connect 函数完成数据包的封装、发送和接收工作以外，其他扫描程序(包括 ping 程序)都需要手动构造数据包。因此，使用原始套接字构造数据包并将其发送到目标主机的指定端口是编写端口扫描器程序的关键。下面将详细讨论使用原始套接字构造并发送数据包的基本流程。

1. 创建原始套接字

socket 函数创建原始套接字的参数 domain 一般选择 AF_INET，表示 IPv4 协议；参数 type 选择 SOCK_RAW，表示建立原始套接字；参数 protocol 表明操作的是哪一种协议的数据包，一般有 IPPROTO_IP、IPPROTO_TCP、IPPROTO_UDP 和 IPPROTO_ICMP 等(代码如下)。

```
Int socket(int domain, int type, int protocol);
```

2. 设置套接字选项

创建原始套接字后，可以根据需要调用 setsockopt 函数来设置当前套接字的选项。setsocketopt 函数的声明如下：

```
int setsockopt(int sockfd, int level, int optname, const char void* optval, socklen_t* optlen);
```

参数 sockfd 是标识套接字的描述符；参数 level 表示控制套接字的层次，对 TCP/IP 协议栈，level 支持 3 种层次 SOL_SOCKET(通用套接字选项)，IPPROTO_IP(IP 选项)和 IPPROTO_TCP(TCP 选项)；参数 Optname 表示 sockfd 所标识的套接字需要设置选项的名称，如果需要构建数据包的 IP 头就将 Optname 设为 IP_HDRINCL；参数 optval 是一个指针，指向存放选项值的缓冲区。optval 需要根据选项不同的数据类型进行转换；参数 optlen 表示 optval 所指向的缓冲区的长度。

3. 创建并填充协议报头

如表 8-1 所示，Linux 系统已经定义了常用协议的报头结构，比如 iphdr、tcphdr、udphdr、icmphdr 等。在创建协议数据包的时候，只需要根据对应的结构体定义申请一块新的内存空间，并用指针指向这块空间的地址即可。结构 iphdr、tcphdr、udphdr、icmphdr 的声明如下。

表 8-1 Linux 系统常用协议的报头结构

协 议 头	数据结构	头 文 件
IP 协议头	struct iphdr	<netinet/ip.h>
TCP 协议头	struct tcphdr	<netinet/tcp.h>
UDP 协议头	struct udphdr	<netinet/udp.h>
ICMP 协议头	struct icmphdr	<netinet/ip_icmp.h>

```
//结构体声明
struct iphdr                                //IP 协议头
{
    #elif defined
    (_LITTLE_ENDIAN_BITFIELD)               //IP 协议的版本定义
    _u8 version :4,
    #elif defined                           //我国一般使用 BIG 的定义
    (_BIG_ENDIAN_BITFIELD)
    _u8 version:4,                          //IP 报头标长
    ihl:4;
    #else
    #error "Please fix"
    #endif                                  //服务类型
    _u8 tos;                                //总长度
    _16 tot_len;                            //标识
    _u16 id;                                //标志+片偏移
    _u16 frag_off;                          //生存时间
    _u8 ttl;                                //协议
    _u8 protocol;                           //头校验和
    _u16 check;                             //源 IP 地址
    _u32 saddr;                             //目的 IP 地址
    _u32 daddr;
};
struct tcphdr                               //TCP 协议头
{
    u_int16_t source;                       //源端口号
    u_int16_t dest;                         //目的端口号
    u_int32_t seq;                          //序号
    u_int32_t ack_seq;                      //确认号
    #if _BYTE_ORDER==_LITTLE_ENDIAN         //LITTLE 版本的定义
```

```
    …
    #elif __BYTE_ORDER == __BIG_ENDIAN
    u_int16_t doff:4;                       //4bit 报头长度
    u_int16_t res1:4;                       //6bit 保留
    u_int16_t res2:2;
    u_int16_t urg:1;                        //UGR 位
    u_int16_t ack:1;                        //ACK 位
    u_int16_t psh:1;                        //PSH 位
    u_int16_t rst:1;                        //RST 位
    u_int16_t syn:1;                        //SYN 位
    u_int16_t fin:1;                        //FIN 位
    #else
    #error "Adjust your defines"
    #endif
    u_int16_t window;                       //窗口大小
    u_int16_t check;                        //校验和
    u_int16_t urg_ptr;                      //紧急指针
};
struct udphdr                               //UDP 协议头
{
    u_int16_t source;                       //源端口
    u_int16_t dest;                         //目的端口
    u_int16_t len;                          //数据报长度
    u_int16_t check;                        //校验和
};
struct icmphdr                              //ICMP 协议头
{
    u_int8_t type;                          //类型
    u_int8_t code;                          //代码
    u_int16_t checksum;                     //0 校验和
    union
    {
      struct                                //echo 数据报
      {
          u_int16_t id;
          u_int16_t sequence;
      } echo;
      u_int32_t gateway;                    //网关地址
      struct                                //MTU 值
      {
          u_int16_t __unused;
          u_int16_t mtu;
      } frag;
    } un;
};
```

4. 发送数据包

在填充完数据包之后，需要调用 sendto 函数将数据包发送出去。函数声明如下，参数 s 是标识套接字的描述符，参数 buf 为指向发送数据包的指针，参数 len 表示内存空间的大小，参数 to 为指向接收方地址的指针，参数 tolen 为指向接收方地址大小的指针。sendto 函数既可用于无连接套接字的通信，也可用于面向连接的套接字通信。当用于面向连接的通信时（套接字类型为 SOCK_STREAM），将省略参数 to 和 tolen。

```
ssize_t sendto(int s, const void* buf, size_t len, int flags, const struct sockaddr* to, socklen_t tolen)
```

需要指出的是，在一般情况下 sendto 函数的 buf 指针所指向的数据包不包含 IP 报头。比如，在 UDP 扫描中，buf 所指向的缓冲区内只包含 UDP 头和数据字段。如果要亲自处理 IP 报头，一定要在第 2 步中调用 setsocketopt 函数设置套接字的选项，代码如下：

```
int flag=1;
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &flag, sizeof(int));
```

调用 setsockopt 函数后，buf 所指向的缓冲区由 IP 头、TCP 头（或者 UDP 头等其他协议头）和数据字段 3 部分组成。

5. 接收数据包

在发送完数据包以后，需要调用 recvfrom 函数来接收响应数据包。recvfrom 函数的声明如下，参数 s 是标识套接字的描述符，参数 buf 为指向接收到信息的指针，参数 len 为内存空间的大小，flags 为控制参数，用于控制是否接收带外数据，参数 from 为指向发送方地址的指针，参数 fromlen 为指向发送方地址大小的指针。同样，recvfrom 函数既可以用于无连接的套接字通信，也可以用于面向连接的通信。

```
ssize_t recvfrom(int s, void* buf, size_t len, int flags, struct sockaddr* from,
                 socklen_t* fromlen)
```

一般在接收数据时，recvfrom 函数处于阻塞状态，直到收到一个数据包才会返回。如果希望 recvfrom 在接收数据包时处于非阻塞状态，则需要调用函数 fcntl 将套接字设置为非阻塞模式。具体代码如下，其中参数 Sockfd 表示所设置的套接字描述符。

```
fcntl(int Sockfd, F_SETFL, O_NONBLOCK)
```

另外，在 Linux 中，可以将套接字看成文件描述符，这样在进行数据发送与接收的时候就可以调用 write 和 read 函数对文件描述符进行读写。write 函数和 read 函数的定义如下：

```
ssize_t write(int fd, const void* buf, size_t nbytes)
ssize_t read(int fd, void* buf, size_t nbyte)
```

write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd 中，若成功，则返回写的字节数；若失败，则返回 −1，并设置 errno 变量。read 函数则负责从 fd 中读取内容。当读取成功时，read 返回实际所读取的字节数；如果返回值是 0，则表示已经读到文件末尾；小于 0，则

表示出现错误。如果错误为 EINTR，则说明读错误是由中断引起的；如果错误是 ECONNREST，则表示网络连接出现问题。

8.3　实例编程练习

8.3.1　编程练习要求

在 Linux 环境中编写一个端口扫描器，利用套接字(socket)正确实现 ping 程序、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描。ping 程序在用户输入被扫描主机 IP 地址之后探测该主机是否可达。其他 4 种扫描在指定被扫描主机 IP 地址、起始端口以及终止端口之后，从起始端口到终止端口对被测主机进行扫描。最后将每一个端口的扫描结果正确地显示出来。

1. 程序输入格式

程序为命令行程序，可执行文件名为 Scaner，命令行格式如下：

```
./Scaner  [选项]
```

其中[选项]是程序为用户提供的多种功能。本程序中，[选项]包括{-h,-c,-s,-f,-u} 5 项基本功能。-h 表示显示帮助信息；-c 表示进行 TCP connect 扫描；-s 表示进行 TCP SYN 扫描；-f 表示进行 TCP FIN 扫描；-u 表示进行 UDP 扫描。

2. 程序的执行过程

(1) 停用 iptables 服务

首先在 Shell 命令行下输入“service iptables stop”停止 iptables 防火墙的过滤功能，以保证端口扫描程序能够正常地接收各种响应数据包。

(2) 打印帮助信息

在控制台命令行中输入./Scaner -h，打印端口扫描器程序的帮助信息(如下所示)。帮助信息详细地说明了程序的各个选项和参数。用户可以通过查询帮助信息充分了解程序的各项功能。

```
[root@ localhost Scaner]# ./Scaner -h
Scaner: usage: [-h]        --help information
               [-c]        --TCP connect scan
               [-s]        --TCP syn scan
               [-f]        --TCP fin scan
               [-u]        --UDP scan
```

(3) 进行 TCP connect 扫描

在控制台命令行中输入./Scaner -c，开始进行 TCP connect 扫描。程序提示用户输入扫描目标主机的 IP 地址，扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@ localhost Scaner]# ./Scaner -c
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully!
Begin TCP connect scan...
Host: 192.168.1.158 Port: 1 closed!
Host: 192.168.1.158 Port: 2 closed!
...
```

(4) 进行 TCP SYN 扫描

在控制台命令行中输入./Scaner -s,开始 TCP SYN 扫描(如下所示)。程序提示用户输入扫描目标主机的 IP 地址、扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后,程序首先利用 ping 程序探测目标主机的 IP 地址是否可达,如果不可达,则放弃对该主机的扫描;否则开启扫描线程,依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@ localhost Scaner]# ./Scaner -s
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully!
Begin TCP SYN scan...
Host: 192.168.1.158 Port: 1 closed!
Host: 192.168.1.158 Port: 2 closed!
...
```

(5) 进行 TCP FIN 扫描

在控制台命令行中输入./Scaner -f,开始进行 TCP FIN 扫描(如下所示)。程序提示用户输入扫描目标主机的 IP 地址、扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后,程序首先利用 ping 程序探测目标主机的 IP 地址是否可达,如果不可达,则放弃对该主机的扫描;否则开启扫描线程,依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@ localhost Scaner]# ./Scaner -f
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully!
Begin TCP FIN scan...
Host: 192.168.1.158 Port: 1 closed!
Host: 192.168.1.158 Port: 2 closed!
```

…

(6) 进行 TCP UDP 扫描

在控制台命令行中输入./Scaner -u，开始进行 UDP 扫描(如下所示)。程序提示用户输入扫描目标主机的 IP 地址、扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@ localhost Scaner]# ./Scaner -u
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:10
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully!
Begin UDP scan...
Host: 192.168.1.158 Port: 1 closed!
Host: 192.168.1.158 Port: 2 closed!
...
```

综上所述，本程序在 Linux 平台下，利用套接字编程实现了 ping 程序，并且在指定扫描端口范围的情况下，实现了对目标主机的 TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描。

8.3.2　编程训练设计与分析

1. 端口扫描器程序 Scaner

端口扫描器程序 Scaner 的流程比较简单。在 main 函数中只需完成与用户进行交互、检测用户输入，以及调用各个扫描功能模块这 3 方面的工作。图 8-1 给出了端口扫描器程序 Scaner 的流程。Scaner 程序的流程分为以下 8 个步骤：

(1) 首先判断是否需要输出帮助信息，若是，则输出端口扫描器程序的帮助信息，然后退出；否则，继续执行下面的步骤。

(2) 用户输入被扫描主机的 IP 地址、扫描起始端口和终止端口。

(3) 判断 IP 地址与端口号是否错误，若错误，则提示用户并退出；否则，继续下面的步骤。

(4) 调用 Ping 函数，判断被扫描主机是否可达，若不可达，则提示用户并退出；否则，继续下面的步骤。

(5) 判断是否进行 TCP connect 扫描，若是，则开启 TCP connect 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来；否则，继续执行下面的步骤。

(6) 判断是否进行 TCP SYN 扫描，若是，则开启 TCP SYN 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来；否则，继续执行下面的步骤。

(7) 判断是否进行 TCP FIN 扫描，若是，则开启 TCP FIN 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来；否则，继续执行下面的步骤。

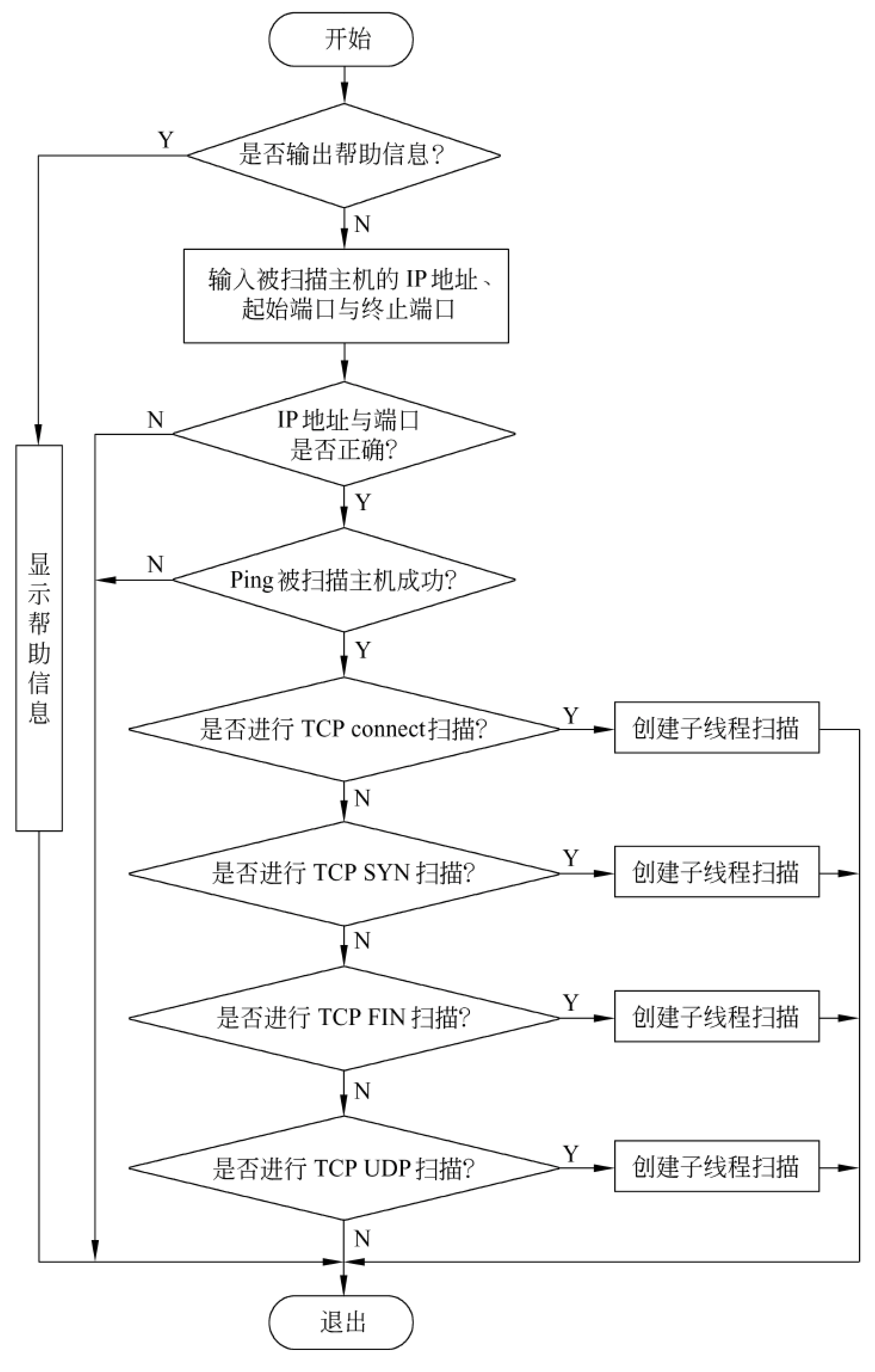


图 8-1 Scaner 程序流程图

(8) 判断是否进行 UDP 扫描，若是，则开启 UDP 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来；否则，继续执行下面的步骤。

(9) 等待所有扫描子线程返回后退出。

在端口扫描器的主流程中先后调用 ping 程序、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描等 5 个功能模块。

2. ICMP 探测指定主机

Ping 程序用于测量本地主机与目标主机之间的网络通信情况。Ping 程序首先发送一个 ICMP 请求数据包给目标主机。如果目标主机返回一个 ICMP 响应数据包，则表示两台

主机之间的通信状况良好，可以继续后面的扫描操作；否则，整个程序将退出。

在端口扫描器程序中，Ping 功能由函数 bool Ping(string HostIP, unsigned LocalHostIP)实现。在 Ping 函数中完成了填充 ICMP 数据包，向目标主机发送请求以及接收响应等功能。若目标主机可达，则返回 true；否则，返回 false。Ping 函数的部分代码如下：

```
bool Ping(string HostIP,unsigned LocalHostIP)
{
    //变量定义
    //创建套接字
    PingSock=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
    ...
    //设置套接字选项
    on=1;
    ret=setsockopt(PingSock,0,IP_HDRINCL,&on,sizeof(on));
    ...
    //创建 ICMP 请求数据包
    SendBufSize=sizeof(struct iphdr)+sizeof(struct icmphdr)+sizeof(struct timeval);
    SendBuf=(char*)malloc(SendBufSize);
    memset(SendBuf,0,sizeof(SendBuf));

    //填充 IP 头
    ip=(struct iphdr*)SendBuf;
    ip->ihl=5;
    ip->version=4;
    ip->tos=0;
    ip->tot_len=htons(SendBufSize);
    ip->id=rand();
    ip->ttl=64;
    ip->frag_off=0x40;
    ip->protocol=IPPROTO_ICMP;
    ip->check=0;
    ip->saddr=LocalHostIP;
    ip->daddr=inet_addr(&HostIP[0]);

    //填充 icmp 头
    icmp=(struct icmphdr*)(ip+1);
    icmp->type=ICMP_ECHO;
    icmp->code=0;
    icmp->un.echo.id=htons(LocalPort);
    icmp->un.echo.sequence=0;

    tp=(struct timeval*)&SendBuf[28];
    gettimeofday(tp,NULL);
    icmp->checksum=in_cksum((u_short*)icmp,
```

```
                    sizeof(struct icmphdr)+sizeof(struct timeval));

//设置套接字的发送地址
PingHostAddr.sin_family=AF_INET;
PingHostAddr.sin_addr.s_addr=inet_addr(&HostIP[0]);
Addrlen=sizeof(struct sockaddr_in);

//发送 ICMP 请求
ret=sendto(PingSock,SendBuf,SendBufSize,0,(struct sockaddr*)&PingHostAddr,
        sizeof(PingHostAddr));
…
//将套接字设置为非阻塞模式
if(fcntl(PingSock, F_SETFL, O_NONBLOCK)==-1)
…
//循环等待接收 ICMP 响应
gettimeofday(&TpStart,NULL);                    //获得循环起始时刻
flags=false;
do
{
    //接收 ICMP 响应
    ret=recvfrom(PingSock,RecvBuf,1024,0,(struct sockaddr*)&FromAddr,
              (socklen_t*)&Addrlen);
    if (ret>0)                                  //如果接收到一个数据包,对其进行解析
    {
        Recvip=(struct ip*)RecvBuf;
        Recvicmp=(struct icmp*)(RecvBuf+(Recvip->ip_hl*4));

        SrcIP=inet_ntoa(Recvip->ip_src);        //获得响应数据包 IP 头的源地址
        DstIP=inet_ntoa(Recvip->ip_dst);        //获得响应数据包 IP 头的目的地址

        in_LocalhostIP.s_addr=LocalHostIP;
        LocalIP=inet_ntoa(in_LocalhostIP);      //获得本机 IP 地址
        //判断该数据包的源地址是否等于被测主机的 IP 地址,目的地址是否等于
        //本机 IP 地址,ICMP 头的 type 字段是否为 ICMP_ECHOREPLY
        if (SrcIP==HostIP && DstIP==LocalIP &&
            Recvicmp->icmp_type==ICMP_ECHOREPLY)
        {   //ping 成功,退出循环   }
    }
    //获得当前时刻,判断等待相应时间是否超过 3 秒,若是,则退出等待
    gettimeofday(&TpEnd,NULL);
        TimeUse=(1000000*(TpEnd.tv_sec-TpStart.tv_sec)+
                (TpEnd.tv_usec-TpStart.tv_usec))/1000000.0;
        if(TimeUse<3)
        …
}while(true);
```

```
    ...
}
```

如图 8-2 所示，Ping 函数的流程可分为以下 8 个步骤：

(1) 为 Ping 程序创建原始套接字(SOCK_RAW)PingSock。

(2) 调用函数 setsockopt，设置套接字选项，使其能够重新构造 IP 数据报头部。

(3) 创建 ICMP 请求数据包。该数据包由 3 部分组成：IP 头、ICMP 头以及数据字段。在本程序中以当前的时间作为数据字段的内容。

(4) 设置套接字 PingSock 的目标主机地址。

(5) 调用 sendto 函数，发送 ICMP 请求数据包。

(6) 调用函数 fcntl 将套接字设置为非阻塞模式。

(7) 调用函数 recvfrom，循环等待接收 ICMP 响应数据包。如果接收到一个数据包，判断 a)源地址是否等于目标主机的 IP 地址，b)目的地址是否等于本机的 IP 地址，c)ICMP 头的 type 字段是否为 ICMP_ECHOREPLY。若上述 3 个条件均满足，则表示成功接收到目标主机发来的 ICMP 响应数据包，Ping 函数返回 true；否则，继续等待，直到超时退出。

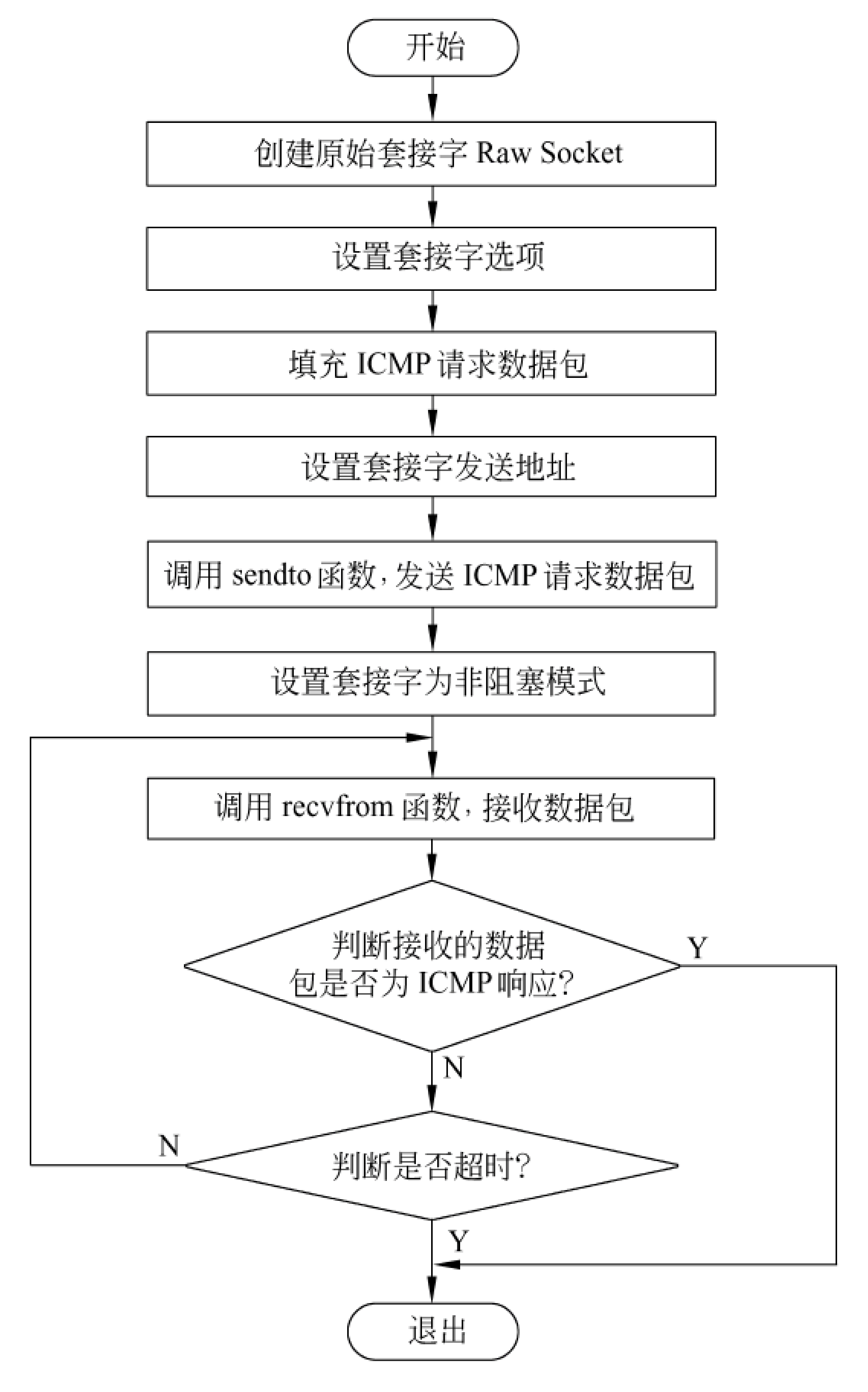


图 8-2　Ping 函数流程图

(8) 如果循环等待时间超过3秒,则退出等待,Ping函数返回false。

3. TCP Connect 扫描

TCP Connect扫描是通过调用流套接字(SOCK_STREAM)的connect函数实现的。该函数尝试连接被测主机的指定端口,若连接成功,则表示端口开启;否则,表示端口关闭。在编程中为了提高效率,采用创建子线程同时扫描目标主机多个端口的方法。线程函数的部分代码如下:

```
void* Thread_TCPconnectHost(void* param)
{
    //变量定义
    //获得目标主机的 IP 地址和扫描端口号
    p=(struct TCPConHostThrParam*)param;
    HostIP=p->HostIP;
    HostPort=p->HostPort;

    //创建流套接字
    ConSock=socket(AF_INET,SOCK_STREAM,0);
    …
    //设置连接主机地址
    memset(&HostAddr,0,sizeof(HostAddr));
    HostAddr.sin_family=AF_INET;
    HostAddr.sin_addr.s_addr=inet_addr(&HostIP[0]);
    HostAddr.sin_port=htons(HostPort);

    //connect 目标主机
    ret=connect(ConSock,(struct sockaddr*)&HostAddr,sizeof(HostAddr));
    if(ret==-1)
    {  //连接失败,端口关闭  }
    else
    {  //连接成功,端口开启  }

    //退出线程
    …
    close(ConSock);                          //关闭套接字
    //子线程数减 1
    pthread_mutex_lock(&TCPConScanlocker);
        TCPConThrdNum--;
    pthread_mutex_unlock(&TCPConScanlocker);
}
//==============================================================
void* Thread_TCPconnectScan(void* param)
{
    //变量定义
    //获得扫描的目标主机 IP、起始端口及终止端口
```

```
    p=(struct TCPConThrParam*)param;
    HostIP=p->HostIP;
    BeginPort=p->BeginPort;
    EndPort=p->EndPort;

    TCPConThrdNum=0;                                //将线程数设为 0
    //开始从起始端口到终止端口循环扫描目标主机的端口
    for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
    {
        //设置子线程参数
        TCPConHostThrParam* pConHostParam=new TCPConHostThrParam;
        pConHostParam->HostIP=HostIP;
        pConHostParam->HostPort=TempPort;

        //将子线程设为分离状态
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

        //创建 connect 目标主机指定的端口子线程
        ret=pthread_create(&subThreadID,&attr,Thread_TCPconnectHost,
                           pConHostParam);
        ...
        //线程数加 1
        pthread_mutex_lock(&TCPConScanlocker);
        TCPConThrdNum++;
        pthread_mutex_unlock(&TCPConScanlocker);
        //如果子线程数大于 100,暂时休眠
        while (TCPConThrdNum>100)
        { sleep(3); }
    }

    //等待子线程数为 0,返回
    while (TCPConThrdNum!=0)
    { sleep(1); }
    //返回主流程
    pthread_exit(NULL);
}
```

上述代码中包含两个线程函数：Thread_TCPconnectScan 和 Thread_TCPconnectHost。这两个函数共同完成 TCP Connect 的扫描工作。其中 Thread_TCPconnectScan 是扫描的主线程函数，该函数在扫描器的主流程中被调用，用于遍历目标主机的端口，创建负责扫描某一固定端口的子线程。而连接(connect)目标主机指定端口的工作由线程函数 Thread_TCPconnectHost 来完成。端口扫描器主流程，函数 Thread_TCPconnectScan 和函数 Thread_TCPconnectHost 3 者之间的关系如图 8-3 所示。为了维护系统中的线程数目，使用变量 TCPConThrdNum 来记录已经创建的子线程数。在函数 Thread_TCPconnectScan

中,每创建一个连接指定端口的子线程,就将 TCPConThrdNum 加 1;而在函数 Thread_TCPconnectHost 退出当前线程之前,将 TCPConThrdNum 减 1。若子线程数大于 100,线程 Thread_TCPconnectScan 就会暂时休眠,等待子线程数降低后再继续工作。当子线程数等于 0 时,表示所有的子线程都已经返回,线程 Thread_TCPconnectScan 就返回程序的主流程。为了保证多个不同线程对子线程数 TCPConThrdNum 的互斥访问,在修改 TCPConThrdNum 之前需要加互斥锁 TCPConScanlocker,修改完毕后再解锁。这样就保证了 TCPConThrdNum 的正确性。

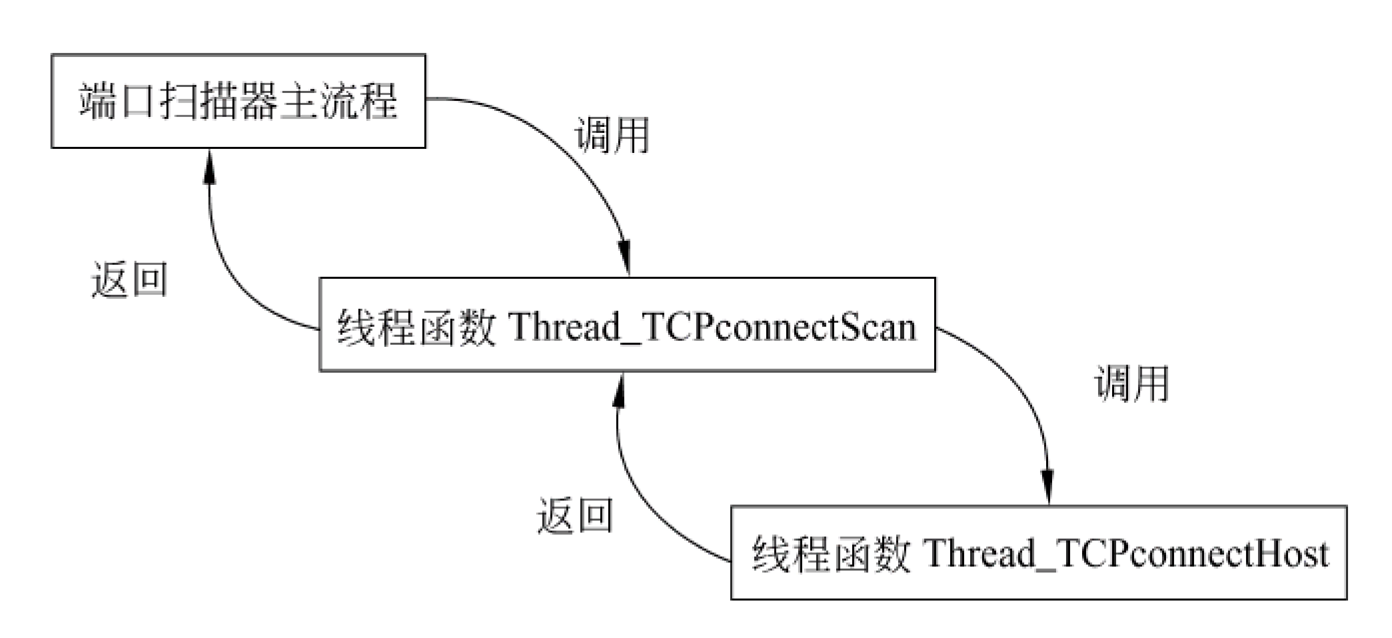


图 8-3 主流程 Thread_TCPconnectScan 和 Thread_TCPconnectHost 之间的关系示意图

线程函数 Thread_TCPconnectHost 是整个 TCP Connect 扫描的核心部分。线程函数 Thread_TCPconnectScan 通过调用它来创建连接目标主机指定端口的子线程。函数 Thread_TCPconnectHost 的工作流程如下:

(1) 获得线程函数 Thread_TCPconnectScan 传来的目标主机的 IP 地址和扫描端口号。

(2) 创建流套接字 ConSock。

(3) 设置连接目标主机的套接字地址 HostAddr。

(4) 调用 connect 函数连接目标主机。若函数返回−1,则表示连接失败;否则,表示连接成功。

(5) 关闭套接字 ConSock,子线程数 TCPConThrdNum 减 1,退出线程。

4. TCP SYN 扫描

本章要求通过原始套接字(SOCK_RAW)来实现 TCP SYN 扫描。原始套接字允许程序员构造数据包的 IP 头字段和 TCP 头字段。虽然 Windows XP 系统出于安全的原因禁止原始套接字发送数据,但是在 Linux 网络编程中,它还是带来了很多方便。

和 TCP Connect 扫描一样,TCP SYN 扫描也包含两个线程函数:Thread_TCPSynScan 和 Thread_TCPSYNHost。Thread_TCPSynScan 是主线程函数,负责遍历目标主机的被测端口,并调用 Thread_TCPSYNHost 函数创建多个扫描子线程。Thread_TCPSYNHost 函数用于完成对目标主机指定端口的 TCP SYN 扫描。这两个函数的部分代码如下:

```
Void* Thread_TCPSYNHost(void* param)
{
    //变量定义
    //获得目标主机的 IP 地址和扫描端口号,以及本机的 IP 地址和端口
    p= (struct TCPSYNHostThrParam* )param;
```

```
…
//设置 TCP SYN扫描的套接字地址
memset(&SYNScanHostAddr,0,sizeof(SYNScanHostAddr));
SYNScanHostAddr.sin_family=AF_INET;
SYNScanHostAddr.sin_addr.s_addr=inet_addr(&HostIP[0]);
SYNScanHostAddr.sin_port=htons(HostPort);

//创建套接字
SynSock=socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
…
//填充 TCP SYN数据包
struct pseudohdr * ptcph=(struct pseudohdr * )sendbuf;
struct tcphdr * tcph=(struct tcphdr * )(sendbuf+sizeof(struct pseudohdr));

//填充 TCP伪头部,用于计算校验和
ptcph->saddr=LocalHostIP;
ptcph->daddr=inet_addr(&HostIP[0]);
ptcph->useless=0;
ptcph->protocol=IPPROTO_TCP;
ptcph->length=htons(sizeof(struct tcphdr));

//填充 TCP头
tcph->th_sport=htons(LocalPort);
tcph->th_dport=htons(HostPort);
tcph->th_seq=htonl(123456);
tcph->th_ack=0;
tcph->th_x2=0;
tcph->th_off=5;
tcph->th_flags=TH_SYN;                              //TCP头 flags字段的 SYN位置 1
tcph->th_win=htons(65535);
tcph->th_sum=0;
tcph->th_urp=0;
tcph->th_sum=in_cksum((unsigned short * )ptcph, 20+12);

//发送 TCP SYN数据包
len=sendto(SynSock, tcph, 20, 0, (struct sockaddr * )&SYNScanHostAddr,
            sizeof(SYNScanHostAddr));
…
//接收目标主机的 TCP响应数据包
len=read(SynSock, recvbuf, 8192);
if(len<=0)
{  //接收错误  }
else
{    …
    //判断响应数据包的源地址是否等于目标主机地址,目的地址是否等于本机
```

```
            //IP地址,源端口是否等于被扫描端口,目的端口是否等于本机端口号
            if(HostIP==SrcIP && LocalIP==DstIP && SrcPort==HostPort &&
                DstPort==LocalPort)
            {
                if (tcph->th_flags==0x14)           //判断是否为 SYN|ACK 数据包
                {  //端口开启   }
                if (tcph->th_flags==0x12)           //判断是否为 RST 数据包
                {  //端口关闭   }
            }
        }

        //退出子线程
        delete p;
        close(SynSock);

        pthread_mutex_lock(&TCPSynScanlocker);
        TCPSynThrdNum--;
        pthread_mutex_unlock(&TCPSynScanlocker);
}
//=============================================================
void* Thread_TCPSynScan(void* param)
{
        //变量定义
        //获得目标主机的 IP 地址和扫描的起始端口号、终止端口号以及本机的 IP 地址
        p=(struct TCPSYNThrParam*)param;
        …
        //循环遍历扫描端口
        TCPSynThrdNum=0;
        LocalPort=1024;

        for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
        {
            //设置子线程参数
            struct TCPSYNHostThrParam* pTCPSYNHostParam=new TCPSYNHostThrParam;
            pTCPSYNHostParam->HostIP=HostIP;
            pTCPSYNHostParam->HostPort=TempPort;
            pTCPSYNHostParam->LocalPort=TempPort+LocalPort;
            pTCPSYNHostParam->LocalHostIP=LocalHostIP;

            //将子线程设置为分离状态
            pthread_attr_init(&attr);
            pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

            //创建子线程
            ret=pthread_create(&subThreadID,&attr,Thread_TCPSYNHost,
```

```
                    pTCPSYNHostParam);
        ...
        //子线程数加 1
        pthread_mutex_lock(&TCPSynScanlocker);
        TCPSynThrdNum++;
        pthread_mutex_unlock(&TCPSynScanlocker);
        //子线程数大于 100,休眠
        while (TCPSynThrdNum>100)
        {   sleep(3);   }
    }
    //等待所有子线程返回
    while (TCPSynThrdNum!=0)
    {   sleep(1);   }
    //返回主流程
    pthread_exit(NULL);
}
```

线程函数 Thread_TCPSYNHost 是整个 TCP SYN 扫描的核心部分。线程函数 Thread_TCPSynScan 通过调用它来创建子线程,向目标主机的指定端口发送 SYN 数据包,并根据目标主机的响应判断端口的状态。函数 Thread_TCPSYNHost 的工作流程如图 8-4 所示。

(1) 获得线程函数 Thread_TCPSynScan 传来的参数,其中包括目标主机 IP 地址、扫描端口号以及本机 IP 地址和端口号。

(2) 设置 TCP SYN 扫描的套接字地址。

(3) 创建原始套接字 SynSock。

(4) 填充 TCP SYN 数据包。注意将 TCP 头 flags 字段的 SYN 位设置为 1。

(5) 调用 sendto 函数向目标主机的指定端口发送 TCP SYN 数据包。

(6) 调用 read 函数接收目标主机的 TCP 响应数据包。若函数的返回值小于 0,则接收错误;否则,继续判断响应数据包的地址是否有误。如果响应数据包的 a)源地址等于目标主机地址,b)目的地址等于本机的 IP 地址,c)源端口号等于被扫描端口号,d)目的端口号等于本机端口号,那么该数据包就是目标主机被扫描端口返回的响应数据包。若该数据包 flags 字段的 ACK 和 SYN 位均置 1,则表示被扫描端口开启。若 flags 字段的 RST 位置 1,则表示被扫描端口关闭。

(7) 关闭套接字 ConSock,子线程数 TCPSynThrdNum 减 1,退出线程。

需要注意的是,在填充 TCP SYN 数据包的过程中调用了函数 in_cksum 计算校验和。该函数将 TCP 伪头部(12 字节),TCP 报头(20 字节)以及应用层数据(程序中为 0 字节)合在一起作为输入数据,将它们按 16 位进行分组计算校验和。同理,在填充 TCP FIN 数据包和 UDP 数据包时,也需要利用函数 in_cksum 计算校验和,只是输入的数据略有不同。函数 in_cksum 的代码如下:

```
unsigned short in_cksum(unsigned short * ptr, int nbytes)
{
```

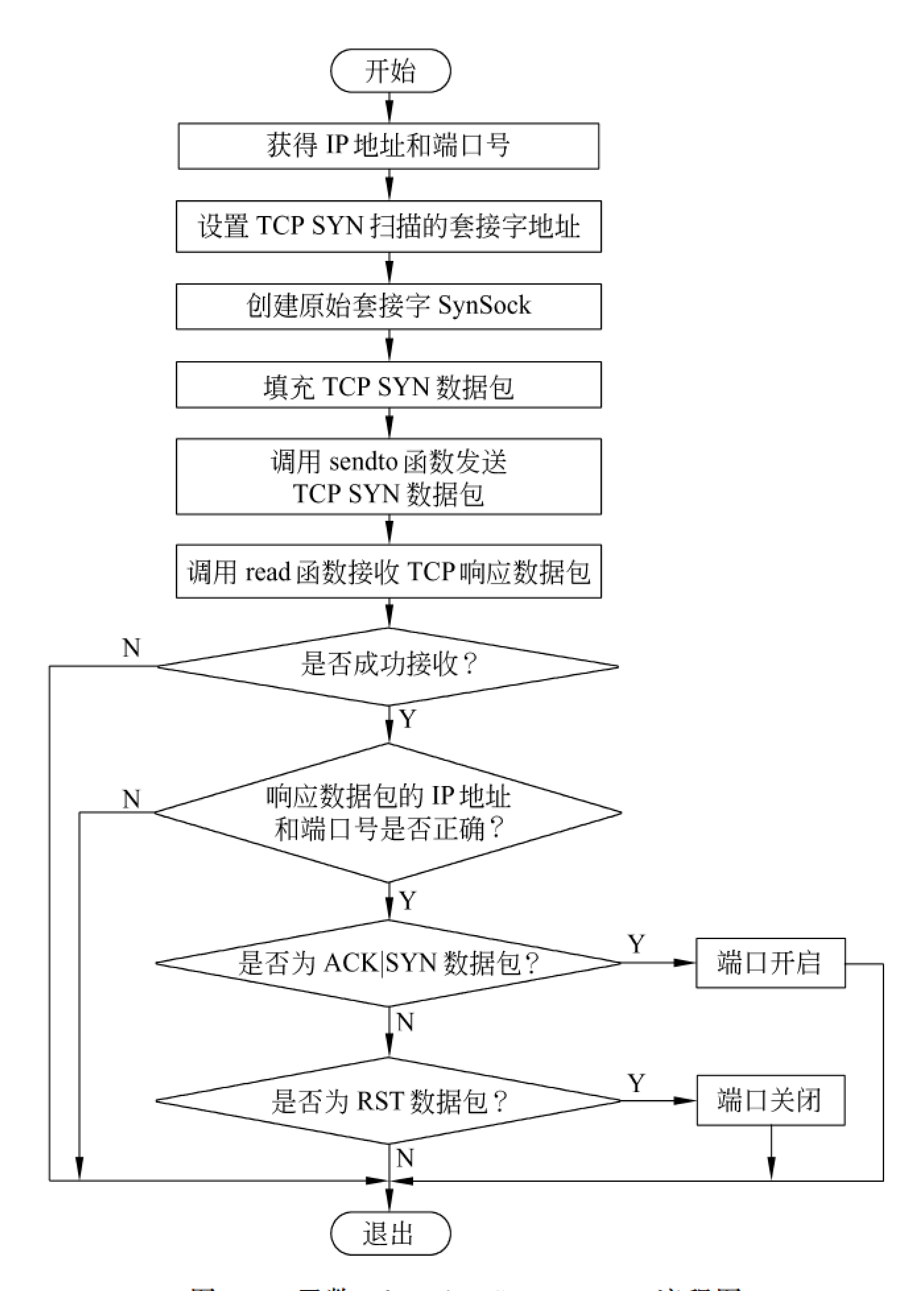


图 8-4 函数 Thread_TCPSYNHost 流程图

```
register long sum;
u_short oddbyte;
register u_short answer;
sum=0;
while(nbytes>1)
 {
     sum+=*ptr++;
     nbytes-=2;
}
if(nbytes==1)
 {
     oddbyte=0;
     *((u_char*)&oddbyte)=*(u_char*)ptr;
     sum+=oddbyte;
}
sum=(sum>>16)+(sum&0xffff);
```

```
    sum+=(sum>>16);
    answer=~sum;
    return(answer);
}
```

5. TCP FIN 扫描

TCP FIN 扫描的代码与 TCP SYN 扫描的代码基本相同。两者都利用原始套接字构造 TCP 数据包发送给目标主机的被测端口。不同之处在于将 TCP 头 flags 字段的 FIN 位置 1。另外在接收 TCP 响应数据包时也略有不同。和前面两种扫描一样,TCP FIN 扫描也包括两个线程函数。它们分别是 Thread_TCPFinScan 和 Thread_TCPFINHost。线程函数 Thread_TCPFinScan 负责遍历端口号,创建 TCP FIN 扫描子线程。它的流程与 TCP SYN 扫描相同。线程函数 Thread_TCPFINHost 的部分代码如下:

```
void* Thread_TCPFINHost(void* param)
{
    //-------------与 TCP SYN 扫描类似-----------------
    …
    //填充 TCP FIN 数据包
    …
    tcph->th_flags=TH_FIN;                //TCP 头 flags 字段的 FIN 位置 1
    …
    //发送 TCP FIN 数据包
    …
    //将套接字设置为非阻塞模式
    if(fcntl(FinRevSock, F_SETFL, O_NONBLOCK)==-1)
    …
    //接收 TCP 响应数据包循环
    gettimeofday(&TpStart,NULL);          //获得开始接收时刻
    do
    {
        //调用 recvfrom 函数接收数据包
        len=recvfrom(FinRevSock,recvbuf,sizeof(recvbuf),0,
        (struct sockaddr*)&FromAddr,(socklen_t*)&FromAddrLen);
        if(len>0)
        {
            SrcIP=inet_ntoa(FromAddr.sin_addr);
            if(SrcIP==HostIP)             //响应数据包的源地址等于目标主机地址
            {
                …
                //判断响应数据包的源地址是否等于目标主机地址,目的地址是
                //否等于本机 IP 地址,源端口是否等于被扫描端口,目的端口是
                //否等于本机端口号
                if(HostIP==SrcIP && LocalIP==DstIP && SrcPort==HostPort &&
                  DstPort==LocalPort)
```

```
                {
                    if (tcph->th_flags==0x14)            //判断是否为 RST 数据包
                    {  …
                        break;
                    }
                }
            }
        }
        //判断等待响应数据包的时间是否超过了 3 秒?
        gettimeofday(&TpEnd,NULL);
        TimeUse= (1000000* (TpEnd.tv_sec- TpStart.tv_sec)+
                (TpEnd.tv_usec- TpStart.tv_usec))/1000000.0;
        if(TimeUse< 3)
          continue;
        else
        {
           //超时,扫描端口开启
           break;
        }
    }
    while(true);

    //退出子线程
    …
}
```

线程函数 Thread_TCPFINHost 的流程与线程函数 Thread_TCPSYNHost 基本相同。在填充 TCP FIN 数据包的时候,注意将 TCP 头的 flags 字段的 FIN 位设置为 1。调用 sendto 函数发送完数据包之后,将套接字 FinRevSock 设置为非阻塞模式。这样 recvfrom 函数就不会一直阻塞,直到接收到一个数据包为止,而是通过一个外部循环控制等待响应数据包的时间。如果超时,则退出循环。在收到一个数据包以后,如果该数据包的 a)源地址等于目标主机地址,b)目的地址等于本机 IP 地址,c)源端口号等于被扫描端口号,d)目的端口号等于本机端口号,则该数据包为响应数据包。如果该数据包 TCP 头的 flags 字段的 RST 位为 1,则表示被扫描端口关闭;否则,继续等待响应数据包。如果等待时间超过 3 秒,则认为被扫描端口是开启的。

6. UDP 扫描

在本章中,UDP 扫描是通过线程函数 Thread_UDPScan 和普通函数 UDPScanHost 实现的。与前面介绍的 TCP 扫描不同,UDP 扫描没有采用创建多个子线程同时扫描多个端口的方式。这是因为目标主机返回的 ICMP 不可达数据包没有包含目标主机的源端口号,扫描器无法判断 ICMP 响应是从哪个端口发出的。因此,如果让多个子线程同时扫描端口,会造成无法区分 ICMP 响应数据包与其对应端口的情况。这样,判断被扫描端口是开启还是关闭就显得毫无意义了。为了保证扫描的准确性,必须牺牲程序的运行效率,逐次地扫描

目标主机的被测端口。

与前面介绍的 TCP 扫描一样，线程函数 Thread_UDPScan 负责遍历目标主机端口，调用函数 UDPScanHost 对指定端口进行扫描，在从起始端口（BeginPort）到终止端口（EndPort）的遍历中，逐次对当前端口（TempPort）进行 UDP 扫描。线程函数 Thread_UDPScan 的部分代码如下：

```
void* Thread_UDPScan(void* param)
{
    …
    //遍历端口，逐次扫描
    for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
    {   …
        UDPScanHost(pUDPScanHostParam);
    }
    …
}
```

函数 UDPScanHost 是 UDP 扫描的核心部分，它负责向被测端口发送 UDP 数据包，并等待接收 ICMP 响应数据包。在发送 UDP 数据包时可以采用两种方法：一种是创建数据报套接字（SOCK_DGRAM）并调用 sendto 函数发送 UDP 数据包；另一种是利用原始套接字（SOCK_RAW）构造一个 UDP 数据包，然后再调用 sendto 函数将该数据包发送给被测端口。本程序采用的是第 2 种方法，部分代码如下：

```
void UDPScanHost(struct UDPScanHostThrParam* p)
{
    //变量定义
    //获得目标主机 IP 地址
    …
    //创建套接字 UDPSock
    UDPSock=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
    …
    //设置套接字 UDPSock 选项
    ret=setsockopt(UDPSock,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));
    …
    //设置 UDP 套接字地址
    memset(&UDPScanHostAddr,0,sizeof(UDPScanHostAddr));
    UDPScanHostAddr.sin_family=AF_INET;
    UDPScanHostAddr.sin_addr.s_addr=inet_addr(&HostIP[0]);
    UDPScanHostAddr.sin_port=htons(HostPort);

    //填充 UDP 数据包
    memset(packet, 0x00, sizeof(packet));
    ip=(struct iphdr*)packet;
    udp=(struct udphdr*)(packet+sizeof(struct iphdr));
    pseudo=(struct pseudohdr*)(packet+sizeof(struct iphdr)-sizeof(struct pseudohdr));
```

```
//填充 UDP 头
udp->source=htons(LocalPort);
udp->dest=htons(HostPort);
udp->len=htons(sizeof(struct udphdr));
udp->check=0;

//填充 UDP 伪头部,用于计算校验和
pseudo->saddr=LocalHostIP;
pseudo->daddr=inet_addr(&HostIP[0]);
pseudo->useless=0;
pseudo->protocol=IPPROTO_UDP;
pseudo->length=udp->len;
udp->check=in_cksum((u_short *)pseudo,
                   sizeof(struct udphdr)+sizeof(struct pseudohdr));

//填充 IP 头
ip->ihl=5;
ip->version=4;
ip->tos=0x10;
ip->tot_len=sizeof(packet);
ip->frag_off=0;
ip->ttl=69;
ip->protocol=IPPROTO_UDP;
ip->check=0;
ip->saddr=LocalHostIP;
ip->daddr=inet_addr(&HostIP[0]);

//发送 UDP 数据包
n=sendto(UDPSock, packet, ip->tot_len, 0,
        (struct sockaddr *)&UDPScanHostAddr, sizeof(UDPScanHostAddr));
...
//设置套接字 UDPSock 为非阻塞模式
if(fcntl(UDPSock, F_SETFL, O_NONBLOCK)==-1)
...
//接收 ICMP 相应数据包循环
gettimeofday(&TpStart,NULL);                  //获得接收起始时间
do
{
    //接收 ICMP 数据包
    n=read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));
    if(n>0)
    {
        //判断 ICMP 数据包的源地址是否等于目标主机地址,code 字段和
        //type 字段的值是否是 3
```

```
            if((hdr.ip.saddr==inet_addr(&HostIP[0])) && (hdr.icmp.code==3) &&
               (hdr.icmp.type==3))
            {
                //UDP 端口关闭
                break;
            }
        }
        //判断等待时间是否超过了 3 秒
        gettimeofday(&TpEnd,NULL);
        TimeUse=(1000000*(TpEnd.tv_sec-TpStart.tv_sec)+
                (TpEnd.tv_usec-TpStart.tv_usec))/1000000.0;
        if(TimeUse<3)
            continue;
        else
        {
            //UDP 端口开启
            break;
        }
    }while(true);

    //关闭套接字
    close(UDPSock);
    delete p;
}
```

如图 8-5 所示，UDPScanHost 函数的流程分为 8 个步骤。

(1) 获得目标主机 IP 地址、扫描端口号以及本机 IP 地址和端口号。

(2) 创建原始套接字 UDPSock。

(3) 设置套接字 UDPSock 的选项。

(4) 设置 UDP 扫描的套接字地址。

(5) 填充 UDP 数据包。注意在填充校验和字段时，需要调用函数 in_cksum 进行计算。

(6) 调用 sendto 函数向目标主机的指定端口发送 UDP 数据包。

(7) 调用 read 函数接收目标主机的 ICMP 响应数据包。若函数的返回值大于 0，则成功接收一个数据包。如果该响应数据包的源地址等于目标主机地址，code 字段和 type 字段的值都为 3，则该数据包就是目标主机被扫描端口返回的 ICMP 不可达数据包。因此，可以认为被扫描的 UDP 端口是关闭的。若接收时间超过 3 秒，则认为被扫描的 UDP 端口开启，退出循环。

(8) 关闭套接字 UDPSock，返回。

7．编写 makefile 文件

本程序包含多个.cpp 文件，在编译和链接时，逐行地输入重复命令会显得十分烦琐。稍有疏忽，就会产生错误。为了解决这个问题，我们可以在这些代码文件的同一目录下创建一个 makefile 文件，每次编译时，只需在 Shell 命令行中输入 make 命令即可。本程序的

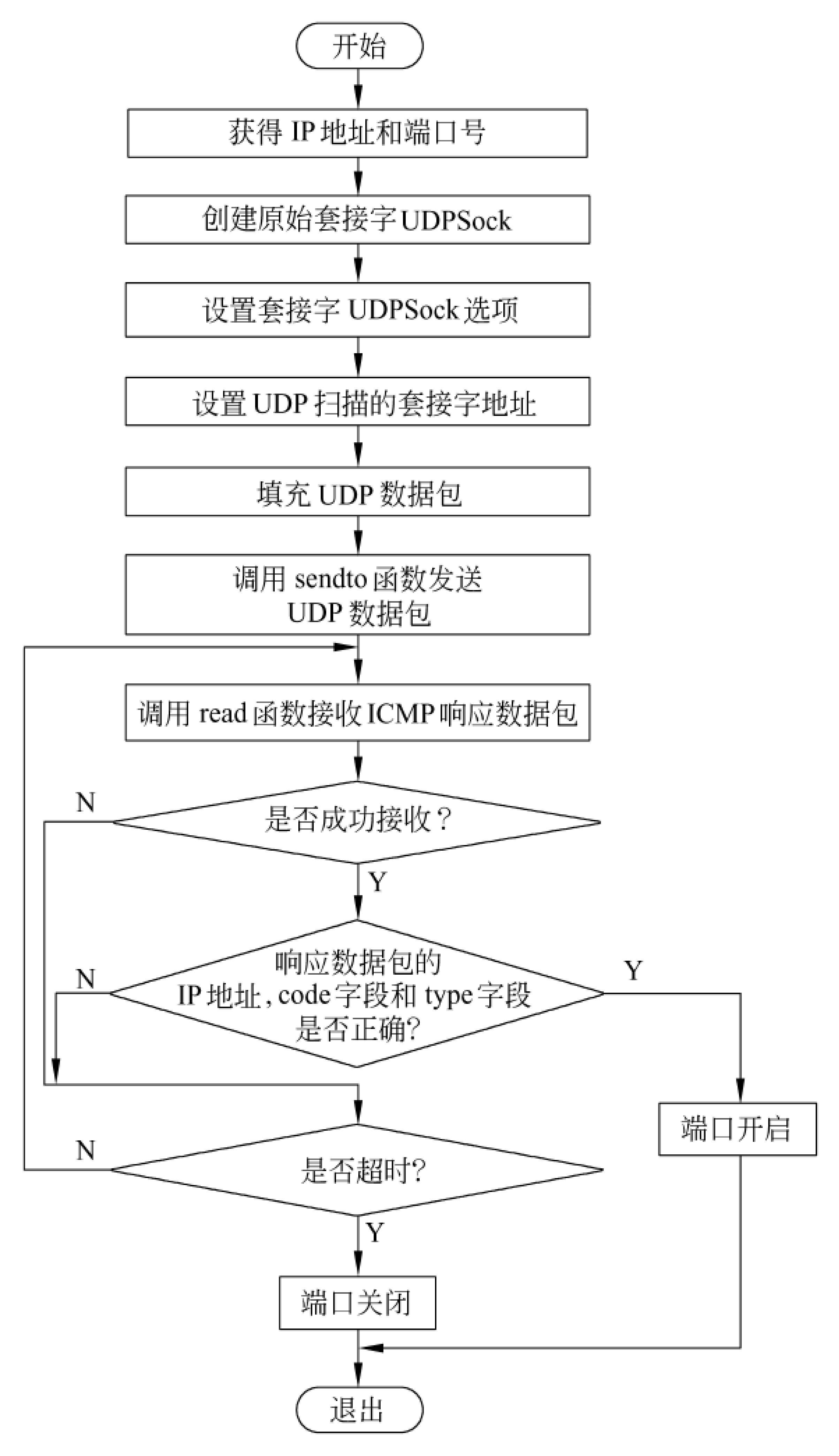


图 8-5　函数 UDPScanHost 的流程图

makefile 文件内容如下所示。由于在本程序中使用了多线程编程技术，所以在最后链接生成可执行代码时应该加上参数-lpthread 才能通过。在 makefile 文件中还使用了 make clean 命令对中间生成的文件进行必要清理，以方便下一次编译。只需在 Shell 命令行中输入 make clean 就可以删除在编译链接时生成的文件。

```
Scaner:Scaner.o TCPConnectScan.o TCPSYNScan.o TCPFINScan.o UDPScan.o
    g++ -lpthread -o Scaner Scaner.o TCPConnectScan.o TCPSYNScan.o TCPFINScan.o UDPScan.o
Scaner.o:Scaner.cpp
    g++ -c Scaner.cpp
TCPConnectScan.o:TCPConnectScan.cpp
    g++ -c TCPConnectScan.cpp
TCPSYNScan.o:TCPSYNScan.cpp
    g++ -c TCPSYNScan.cpp
```

```
TCPFINScan.o:TCPFINScan.cpp
	g++ -c TCPFINScan.cpp
UDPScan.o:UDPScan.cpp
	g++ -c UDPScan.cpp
clean:
	rm Scaner
	rm Scaner.o
	rm TCPConnectScan.o
	rm TCPSYNScan.o
	rm TCPFINScan.o
	rm UDPScan.o
```

8.4　扩展与提高

8.4.1　ICMP 扫描扩展

ICMP 扫描就是利用 8.2.1 小节介绍的 ping 程序判断目标主机是否可达。但是这种传统的 ICMP 扫描方式容易被防火墙过滤。如何才能绕开防火墙探测它后面的主机呢？ICMP 扩展扫描方式就做到了这一点。

ICMP 扩展扫描利用了 ICMP 协议最基本的用途——报错。根据 TCP/IP 协议，如果在通信过程中出现错误，则接收端将产生一个 ICMP 的错误报文，用来通告发送端错误的相关信息。这些错误报文是根据 ICMP 协议要求由探测系统自动产生的，一般不会被防火墙拦截。这也是黑客常用的手段，了解它对于修复安全漏洞非常有益。

下面列举出 ICMP 扩展扫描的 4 种实现方式：

(1) 向目标主机发送一个只有 IP 头的 IP 数据报，目标主机将返回 Destination Unreachable 的 ICMP 错误报文。

(2) 向目标主机发送一个错误的 IP 数据报，比如，IP 头长度错误。目标主机将返回 Parameter Problem 的 ICMP 错误报文。

(3) 当数据报分片，但是却没有给接收端足够的分片时，接收端分片组装超时会发送分片组装超时的 ICMP 错误报文。

(4) 向目标主机发送一个 IP 数据报，但是协议项是错误的，比如协议项不可用，则目标将返回 Destination Unreachable 的 ICMP 错误报文。

此外，扩展 ICMP 扫描还有两种其他的功能：

(1) 探测防火墙的存在：将数据包的 IP 头协议字段填入一个至今无人使用的较大的值，向确定存在的主机发送该数据包，若收不到响应，则说明目标主机有防火墙保护。

(2) 探测目的主机运行协议：因为 IP 头协议字段的长度为 8 位，所以只有 256 种协议的可能。遍历这 256 种可能的协议，探测目标主机，便可穷举出目标主机当前运行的协议。

8.4.2　TCP 扫描扩展

在 8.2.2 小节介绍的 TCP FIN 扫描属于秘密扫描的一种，所谓秘密扫描就是在扫描过程中完全不涉及 TCP 连接建立过程，确保系统不会记录任何日志的扫描技术。

1. 理论基础

(1) 当一个 SYN 或者 FIN 数据包到达一个关闭的端口时,根据 TCP 协议,该端口在丢弃数据包的同时,会发送一个 RST 数据包。

(2) 当一个 RST 数据包到达一个监听端口时,RST 数据包会被丢弃。

(3) 当一个 RST 数据包到达一个关闭端口时,RST 数据包会被丢弃。

(4) 当一个包含 ACK 的数据包到达一个监听端口时,数据包会被丢弃,同时发送一个 RST 数据包。

(5) 当一个 SYN 数据包到达一个监听端口时,正常的三阶段握手继续,回应一个 ACK |SYN 数据包。

(6) 当一个 FIN 数据包到达一个监听端口时,数据包会被丢弃。

(7) 当收到 URG 和 PSH 标志位置位的数据包时,如果端口关闭,则返回 RST 数据包;否则监听端口将丢弃该数据包。所有 URG、PSH、FIN 标志位置位或者无任何标记的 TCP 数据包都会引发监听端口的丢弃行为。

2. 其他扫描方式

(1) ACK 扫描

发送一个只有 ACK 标志的 TCP 数据包给主机,如果主机反馈一个 TCP RST 数据包,则这个主机是存在的。也可以通过这种技术来确定对方防火墙仅仅是简单的分组过滤,还是基于状态的防火墙。

(2) NULL 扫描

发送一个没有任何标志位的 TCP 数据包,根据 RFC793,如果目标主机的相应端口是关闭的,则应该返回一个 RST 数据包。

(3) FIN+URG+PUSH 扫描

向目标主机发送一个 FIN、URG 和 PUSH 分组,根据 RFC793,如果目标主机的相应端口是关闭的,则应该返回一个 RST 数据包。

秘密扫描虽然种类繁多,但是大都依赖操作系统网络协议栈的实现细节。因此,秘密扫描并非对所有操作系统有效。针对不同的操作系统,只有选择合适的扫描方式,才能达到预期的扫描目标。

此外,TCP 扫描还有一种扩展方式叫做间接扫描,即扫描发起主机冒充某台不相关的主机,以其 IP 地址填充扫描所需数据包的源地址,从而实现隐藏自身的目的。此方式的原理比较简单,这里不再赘述。

8.4.3 系统漏洞扫描简介

漏洞扫描最初是以黑客攻击技术出现的。黑客通过自己编写或利用程序来检测待攻击目标是否存在特定的漏洞,以便发动有效地攻击。随着扫描技术的发展和漏洞资料库的逐步完善,进而出现了专业的安全评估工具——漏洞扫描器。

漏洞扫描器通常分为主机型和网络型两大类。主机型漏洞扫描有时也称被动扫描,它采用非破坏性的方法对系统的文件属性,操作系统的安全补丁、账户设置、服务配置以及应

用程序等进行安全检测。由于是以一定的用户权限进行检测的，所以它能够准确地定位系统的问题，发现漏洞。网络型漏洞扫描器通过模拟黑客攻击的方式来进行安全漏洞检测。扫描器先采用定制的脚本模拟攻击系统，然后对结果进行分析，看是否与漏洞库存储的规则匹配。如果匹配，则说明存在安全漏洞。由此可见，扫描器漏洞资料库的完善与否直接影响扫描器对漏洞的检测能力。

目前的漏洞扫描产品大部分都属于网络型，同时它们也吸取了主机型扫描器的优点。它们既可以对网络上的服务器进行远程漏洞检测，又可以对特定主机进行更深层次的安全检测。常见的漏洞扫描产品有 SSS、ISS、eEye、RETINA、NAI CYBERCOP 及 Nmap 等。

网络漏洞扫描器对目标系统进行漏洞检测时，首先探测目标系统的存活主机，对存活主机进行端口扫描，确定系统开放的端口，同时根据协议指纹技术识别出主机的操作系统类型。然后扫描器对开放的端口进行网络服务类型的识别，确定其提供的网络服务。漏洞扫描器根据目标系统的操作系统平台和提供的网络服务，调用漏洞资料库中已知的各种漏洞进行逐一检测，通过对探测响应数据包的分析判断是否存在漏洞。

现有的网络漏洞扫描器主要是利用特征匹配的原理来识别各种已知的漏洞的。扫描器发送含有某一漏洞特征探测码的数据包，根据返回数据包中是否含有该漏洞的响应特征码来判断是否存在漏洞。例如，对于 IIS 中的 Unicode 目录遍历漏洞，扫描器只要发送含有特征代码%c1%1c 的探测包：http://x. x. x. x/scrlpts/.. %c1%1c../winnt/system32/cmdexe?/c+dir，如果应答数据包中含有 200 OK 则可以断定该漏洞存在。

由此可见，漏洞扫描是一项基于漏洞数据库进行特征比对的扫描技术，在系统安全检测方面具有广泛的应用前景。

8.4.4　Linux 环境中 Nmap 的安装与使用

1. Nmap 简介

网络映射器(Network mapper，Nmap)是一种开放源代码的网络探测和安全审计程序。系统管理员与用户不仅可以使用 Namp 快速地扫描大型网络，发现网络中运行的主机，而且可以进一步探测这些主机所能提供的服务，操作系统的相关信息以及报文过滤器或防火墙的类型等。虽然 Nmap 通常用于安全审计，但是许多系统管理员和网络管理员也用它来做一些日常工作，比如查看整个网络的信息，管理服务升级计划以及监视主机和服务的运行等。

Nmap 支持多种协议的扫描，比如 UDP、TCP connect、TCP SYN (half open)、FIN、ACK sweep、ICMP (ping sweep)、ftp proxy (bounce attack)、Reverse-ident、Xmas Tree 和 Null 扫描。此外，Nmap 还提供了一些高级功能，例如通过 TCP/IP 协议栈特征探测操作系统类型，秘密扫描，动态延时和重传计算，并行扫描，通过并行 ping 扫描探测关闭的主机，诱饵扫描，避开端口过滤检测，直接 RPC 扫描(无需端口映射)，碎片扫描以及灵活的目标和端口设定。在 Linux 系统中，非 root 用户可以利用 Nmap 完成许多工作，但是关键的核心功能(比如 raw socket)需要 root 权限才能运行。

2. Nmap 安装

许多操作系统都支持 Nmap 的安装，比如 Linux、Microsoft Windows、Mac OS X、Sun

Solaris 及 OpenBSD 等。根据不同的操作系统，Nmap 提供了多种安装方式。其中，将源代码进行编译并安装是一种传统而有效的安装方式。下面将介绍采用这种方式来安装 Nmap 的过程。

（1）准备工作

如表 8-2 所示，本章选择 Nmap-4.85BETA 版，可以从 http://nmap.org/download.html 下载相关的源代码压缩包。在安装 Nmap 之前，需要检查 ubuntu 是否安装了 g++编译器。如果没有，以 root 身份登录系统，在终端或 shell 命令行中执行命令"apt-get install g++"，系统就会自动地下载并安装 g++编辑器。

表 8-2　Nmap 与操作系统信息

项　目	说　　明	项　目	说　　明
Nmap 安装文件	nmap-4.85BETA8.tar.bz2	操作系统版本	ubuntu-8.10

（2）安装步骤

在完成上述准备工作之后，就可以开始 Nmap 的安装过程。具体操作步骤如下：

① 以 root 身份登录系统，或者输入"su root"转换为 root 用户。

② 在终端或 shell 命令行中执行命令"tar xvjf nmap-4.85BETA8.tar.bz2"，将 Nmap 软件包的内容解压至同一目录下的文件夹 nmap-4.85BETA8 中。

③ 执行命令"cd nmap-4.85BETA8"进入解压后的文件夹。

④ 执行命令"./configure"对当前 Linux 系统进行配置。如果配置成功，在命令行中会显示一条由二进制字符组成的喷火巨龙，如图 8-6 所示。

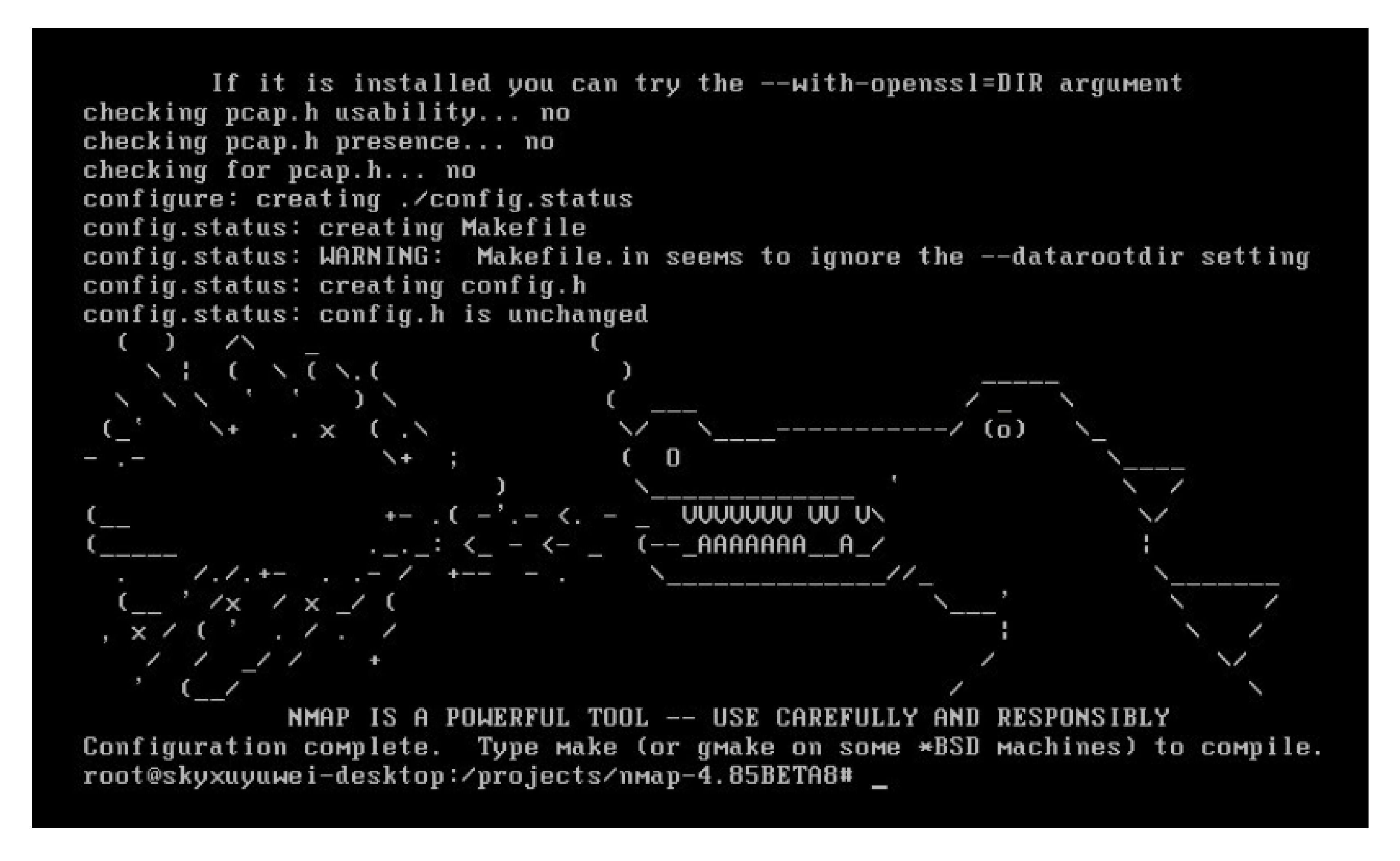


图 8-6　Nmap 配置成功界面图

⑤ 执行命令"make"对 Nmap 进行编译。

⑥ 执行命令"make install"，将 Nmap 安装到/usr/local/bin/nmap 目录下。

3. Nmap 使用

Nmap 的语法如下：

```
nmap [扫描类型] [功能选项] [目标说明]
```

（1）扫描类型

Nmap 支持十几种扫描技术。除了 UDP 扫描(-sU)可以和任何一种 TCP 扫描类型结合使用外，一般一次只使用一种方法进行扫描。扫描类型的格式是-s[*]，其中[*]表示一个字符，表示特定的扫描类型。表 8-3 列出了各种扫描类型所对应的格式。其中，deprecated FTP bounce 扫描(-b)是一个例外。

表 8-3　Nmap 扫描类型列表

扫描类型	说　明	扫描类型	说　明
-sS	TCP SYN 扫描	-sT	TCP connect 扫描
-sU	UDP 扫描	-sN	TCP NULL 扫描
-sF	TCP FIN 扫描	-sX	TCP XmasTree 扫描
-sA	TCP ACK 扫描	-sW	TCP 窗口扫描
-sM	TCP Maimon 扫描	-sO	IP 协议扫描
-b	FTP 弹跳扫描		

（2）功能选项

Nmap 的功能选项可以组合使用。其中一些功能选项只能够在某种特定的扫描模式下使用。Nmap 会自动识别无效或者不支持的功能选项组合，并向用户发出警告信息。因为 Nmap 的功能选项种类繁多，所以本章不再逐一进行详细介绍。读者可以登录到 Nmap 的官方网站查阅相关内容(http://nmap.org/book/man-briefoptions.html)。

（3）目标说明

除扫描类型和功能选项以外，Nmap 命令中剩余的部分都被视为对目标主机的说明。以 TCP SYN 扫描为例，最简单的情况是只指定一个目标 IP 地址或主机名，扫描结果如下所示。

```
root@skyxuyuwei-desktop:~# nmap-sS 192.168.1.1

Starting Nmap 4.85BETA8 ( http://nmap.org ) at 2009-04-27 22:07 CST
Interesting ports on 192.168.1.1:
Not shown: 999 closed ports
PORT    STATE SERVICE
23/tcp open   telnet
MAC Address: 00:E0:FC:04:01:AA (Huawei Technologies CO.)

Nmap done: 1 IP address (1 host up) scanned in 10.92 seconds
```

如果希望扫描整个网络的相邻主机，则目标说明可以采用一个 CIDR 风格的地址。只

要在一个 IP 地址或主机名后面加上“/<numbit>”，Nmap 就会扫描所有与该参考 IP 地址具有 <numbit>位相同比特的所有 IP 地址或主机。<numbit>所允许的最小值是 1，这将会扫描半个互联网；最大值是 32，这将会扫描该主机或 IP 地址。例如，192.168.1.0/24 将会扫描 192.168.1.0 (二进制格式：11000000 10101000 00000001 00000000)和 192.168.10.255(二进制格式：11000000 10101000 00001010 11111111)之间的 256 台主机。以 TCP SYN 扫描为例，扫描结果如下所示。

```
root@skyxuyuwei-desktop:~# nmap -sS 192.168.1.0/24

Starting Nmap 4.85BETA8 ( http://nmap.org ) at 2009-04-27 22:11 CST
Interesting ports on 192.168.1.1:
Not shown: 999 closed ports
PORT STATE SERVICE
23/tcp open   telnet
MAC Address: 00:E0:FC:04:01:AA (Huawei Technologies CO.)

Interesting ports on 192.168.1.23:
Not shown: 996 filtered ports
PORT       STATE   SERVICE
139/tcp    open    netbios-ssn
445/tcp    open    microsoft-ds
912/tcp    open    unknown
2869/tcp   closed  unknown
MAC Address: 00:21:9B:13:4C:0F (Dell)

Interesting ports on 192.168.1.122:
Not shown: 999 filtered ports
PORT    STATE SERVICE
912/tcp open    unknown
MAC Address: 00:24:8C:0C:9F:DD (Unknown)

Interesting ports on 192.168.1.136:
Not shown: 981 closed ports
PORT       STATE   SERVICE
25/tcp     open    smtp
53/tcp     open    domain
80/tcp     open    http
88/tcp     open    kerberos-sec
135/tcp    open    msrpc
139/tcp    open    netbios-ssn
389/tcp    open    ldap
445/tcp    open    microsoft-ds
464/tcp    open    kpasswd5
593/tcp    open    http-rpc-epmap
636/tcp    open    ldapssl
```

```
1025/tcp    open    NFS-or-IIS
1027/tcp    open    IIS
1037/tcp    open    unknown
1038/tcp    open    unknown
1041/tcp    open    unknown
1050/tcp    open    java-or-OTGfileshare
3268/tcp    open    globalcatLDAP
3269/tcp    open    globalcatLDAPssl
MAC Address: 00:0C:29:77:52:83 (VMware)

Interesting ports on 192.168.1.158:
Not shown: 996 filtered ports
PORT        STATE     SERVICE
139/tcp     open      netbios-ssn
445/tcp     open      microsoft-ds
2869/tcp    closed    unknown
3389/tcp open    ms-term-serv
MAC Address: 00:16:76:A9:55:3A (Intel)

Interesting ports on 192.168.1.222:
Not shown: 997 filtered ports
PORT      STATE   SERVICE
139/tcp   open    netbios-ssn
445/tcp   open    microsoft-ds
2869/tcp    closed    unknown
MAC Address: 00:24:8C:0D:58:0B (Unknown)

Interesting ports on 192.168.1.244:
Not shown: 999 closed ports
PORT    STATE SERVICE
22/tcp open    ssh

Nmap done: 255 IP addresses (7 hosts up) scanned in 36.59 seconds
```

CIDR 标志位虽然非常简洁,但有时却显得不够灵活。例如,扫描 192.168.0.0/16,但略过任何以.0 或者.255 结束的 IP 地址(通常为广播地址)。Nmap 通过设定每 8 位 IP 地址的范围支持这种扫描。用户可以用"-"分开的数字或范围列表为 IP 地址的每 8 位组指定范围。例如,192.168.0-255.1-254 将略过在该范围内以.0 和.255 结束的地址。范围设定不限于 IP 地址的最后 8 位。例如,0-255.0-255.13.37 将在整个互联网范围内扫描所有以 13.37 结束的地址。这种大范围的扫描对互联网的调查研究是有益的。

此外,目标说明不仅局限于命令行指定的方式,还可以通过选项-iL 从列表中输入。如果用户希望对互联网中的主机进行探测,则可以通过-iR 选项随机地选择目标主机。

第 9 章 网络诱骗系统设计与实现

9.1 本章训练目的与要求

网络诱骗是主动网络防护与网络取证的主要手段之一，对于保护网络应用系统安全具有重要作用。本章在讨论网络诱骗基本原理的基础上，系统地研究系统结构设计与软件编程的基本方法。

本章训练的主要目的是：

(1) 理解网络诱骗系统的基本工作原理。

(2) 理解 Linux 系统调用实现和原理，以及通过 Hook 技术对操作系统原有 API 加以扩展的方法。

(3) 掌握 Loadable Kernel Module 编程的相关知识和方法。

(4) 了解 Linux 系统中程序隐藏的方法。

本章的训练内容是设计并实现一种简单的网络诱骗系统(Honey Pot)，主要要求如下：

(1) 运行于 Linux 平台，工作在系统核心层(Ring 0)。

(2) 具有记录用户终端登陆后键盘输入的能力，并将记录的信息以日志形式存储。

(3) 有能力的读者可以尝试添加网络诱骗系统的隐藏功能(模块、文件及通信等)。

9.2 相关背景知识

9.2.1 网络诱骗系统的技术手段

网络诱骗系统是用来观测、记录攻击者探测及入侵系统行为特征的一类网络安全软件。网络诱骗系统一般需要存储一些对于攻击者具有较大诱惑力的数据或应用程序作为诱饵，同时通过一些特殊配置来诱惑潜在的攻击者进行攻击，并对其攻击行为进行监控、记录，进而评估其危害，掌握攻击的证据，达到主动保护系统和网络，为依法惩治攻击者提供必要的依据。网络诱骗系统的主要技术包括伪装技术、监控技术以及隐藏技术。

1. 伪装技术

伪装技术用于在网络诱骗系统上虚拟特定安全漏洞或者网络服务，用以吸引攻击者进行攻击，其主要包括已知漏洞伪装和服务伪装两种。

(1) 已知漏洞伪装

为了使网络诱骗系统对入侵者更有吸引力，需要采用各种欺骗手段。例如在欺骗主机

上模拟一些操作系统，一些网络攻击者最“喜欢”的端口和各种存在入侵可能的漏洞特征等。

a. 模拟存在注入漏洞的 Web 服务器

一般的注入入侵者都使用在页面链接后添加“1＝1”和“1＝2”的方式探测注入漏洞，可以在网页中构造如下代码，模拟存在注入漏洞的网页，从而欺骗入侵者。

```
<%
Id=Request('id');
sid=right(id,1);
uid=right(id,3);
if(sid="'" then Response("SQL语句错误<br>SQL语句后有未闭合的引号")
elseif sid=";"then Response("SQL语句错误<br>SQL语句操作符丢失")
end if
if uid="1=1"then Response("<iframe src='list.asp?id=1"width=100%height=100%frameborader=0></iframe>")
elseif uid="1=2"then Response("没有找到相关数据")
else Response ("SQL语句错误<br>SQL语法错误!")
end if
%>
```

上述代码模拟了存在注入漏洞的网页在接受探测时的表现行为，当然，为了提升模拟效果，可以使用正则表达式代替简单的字符比对，从而增加欺骗的普适性。

b. 模拟 IIS Unicode 目录遍历漏洞

攻击者使用网络漏洞扫描器对目标系统进行漏洞检测时，首先探测目标系统的存活主机，对存活主机进行端口扫描，确定系统开放的端口，同时根据协议指纹技术识别出主机的操作系统类型。然后扫描器对开放的端口进行网络服务类型的识别，确定其提供的网络服务。漏洞扫描器根据目标系统的操作系统平台和提供的网络服务，调用漏洞资料库中已知的各种漏洞进行逐一检测，通过对探测响应数据包的分析判断是否存在漏洞。

现有的网络漏洞扫描器主要是利用特征匹配的原理来识别各种已知的漏洞的。扫描器发送含有某一漏洞特征探测码的数据包，根据返回数据包中是否含有该漏洞的响应特征码来判断是否存在漏洞。例如，对于 IIS 中的 Unicode 目录遍历漏洞，扫描器只要发送含有特征代码%c1%1c 的探测包：http://x. x. x. x/scrlpts/..%c1%1c../winnt/system32/cmd exe?/c+dir，如果应答数据包中含有字符串“200 OK”则可以断定该漏洞存在。所以可以伪造一个 http 服务，模拟上述漏洞的反应，进而吸引攻击者进行攻击。

(2) 服务伪装

使用端口重定向技术可以在网络诱骗系统中模拟一个工作于其他主机的系统服务。这样可以在引诱攻击者进行攻击的同时，不对真正的服务运行主机造成任何伤害。

例如，ServerU 权限提升漏洞是由于 ServerU 保存用户配置文件时，未对其进行保护，进而导致一个具有特定目录写权限的用户可以通过覆盖用户配置文件来获得执行权限。通过端口重定向技术就可以在避免目的主机入侵的前提下安全地观测攻击者的入侵过程，具体配置方法如下。

假定存在主机 A 为正常的 FTP 服务器，主机 B 为网络诱骗系统，其可以开放自身的 21 端口，模拟为 FTP 服务器，当主机 B 接收到其他主机连接其 21 端口的请求后，马上创建一个新的套接字，并连接主机 A 的 21 端口，并在外来主机与主机 A 之间进行数据中转和监

控。针对数据连接的操作与之类似。与此同时,主机 B 需要对传输的 FTP 命令进行过滤,防止其对主机 A 产生破坏,在本例中主机 B 需要过滤全部执行(exec)命令。

这样在攻击者进行攻击时,主机 B 就会完整记录下攻击者的攻击手段,记录通过程序漏洞提升权限的全部过程,同时阻止攻击者的进攻行为对主机 A 造成的伤害。攻击者希望运行的程序一般为攻击者上传的木马等恶意程序。

2. 监控技术

网络诱骗系统的主要功能是对攻击者的行为进行监控,为实现这个目标需要以下两方面的功能:

(1) 数据控制

数据控制的目的是确保网络诱骗系统中的诱骗主机不会被用来作为攻击网络中的其他非诱骗主机的跳板。网络诱骗系统的数据控制必须保证不被攻击者发现,否则会影响系统的诱骗效果。上文提到的在通过端口重定向模拟 FTP 服务器过程中对"exec"命令的过滤就属于数据控制的范畴。此外,数据控制技术还包括禁止本机开放新端口,禁止对外连接等。

(2) 信息捕获

数据捕获的目的是在尽可能隐蔽的情况下记录攻击者攻击行为的特征数据,包括击键序列及向网络中发送的数据包等信息。攻击行为特征数据记录的越全面,越容易对其特点进行分析以找到相应的应对措施。

需要捕获的信息包括攻击者上传的文件内容,键盘的输入命令和网络活动等,在不同平台下有不同的实现方式:例如在 Win32 平台下,可以通过注册键盘钩子回调函数在键盘操作的时候通过回调函数获得键盘的输入信息,也可以通过编写键盘过滤驱动直接在 Ring0 获取键盘的输入内容,针对文件监控,可以使用磁盘过滤驱动监控文件的写入,使用 SPI 技术或者 Hook NDIS 分别从应用层和链路层对网络活动进行监控;而在 Linux 下可以通过注册自己的中断响应句柄或者劫持系统函数对键盘输入进行监控,可供选择的系统函数包括 handle_scancode()、put_queue()、receive_buf()、tty_read()以及 sys_read(),本章的示例程序就是通过劫持 sys_read()函数实现的。至于网络监控方面,Linux 提供了 EB table 和 IP table 可以分别从链路层和网络层对网络数据进行监控。由于 Linux 是一款开源的操作系统,所以可以通过更改系统源代码对原有系统功能进行扩展,从而实现对信息的监控和捕获。

3. 隐藏技术

在攻击者入侵的同时,网络诱骗系统将记录攻击者的输入、输出信息,键盘记录信息,屏幕信息以及攻击者曾使用过的工具,并分析攻击者所要进行的下一步行为。捕获的数据不能直接放在网络诱骗系统主机上,因为有可能被攻击者发现,从而使其觉察到这是一个"陷阱"而提早退出。所以,可以通过一些 Root kit 技巧对文件进行隐藏,或者使用网络协议将数据直接发送到专用的日志服务器进行数据记录。以开源项目 Sebek 网络诱骗系统为例,其使用 UDP 协议直接将记录的日志数据发送到日志服务器,为了防止攻击者监控其通信数据,Sebek 跳过系统协议栈,手动构造 UDP 头、IP 头以及链路层包头,并直接使用网卡驱

动接口发送数据包。此外 Sebek 还拦截了系统 Raw Socket 的调用，从而确保攻击者使用 Sniffer 也无法发现其日志数据。

此外，网络诱骗系统本身也可能被攻击者发现。所以，必须通过进程隐藏、模块隐藏及文件隐藏等相关技巧对网络诱骗系统本身进行隐藏，以增加系统的安全性。本章扩展提高部分就对如何在 Linux 系统中隐藏内核模块、文件及通信端口进行了介绍。

9.2.2 网络诱骗系统分类

网络诱骗系统可以根据以下 3 种情况进行分类。

1. 网络诱骗系统配置的复杂性分类

按照网络诱骗系统配置的复杂性，网络诱骗系统可以分为单机网络诱骗系统和网络诱骗系统两类：

(1) 单机网络诱骗系统一般使用一台主机作为目标系统的副本，同真实的系统安装同样的操作系统，提供同样的服务。这种网络诱骗系统中除了有很多已知的系统漏洞外，一般还存在一些诱人的虚假信息(如公司的财务报表、一些重要的客户资料等)用于引诱攻击者进行攻击。

(2) 网络诱骗系统在多个单机网络诱骗系统的外围加入一些传统的网络安全防御措施，进而对进出单机网络诱骗系统的数据流量进行控制，防止其被攻击者作为攻击其他非网络诱骗系统的跳板。由于单机网络诱骗系统中的数据直接进入网络，所以仅靠单机网络诱骗系统难以控制外出的数据流量，系统很有可能被攻击者用作攻击网络上的其他非网络诱骗系统，所以在实际的应用中，单机网络诱骗系统很少出现。

2. 网络诱骗系统部署的目标分类

按照网络诱骗系统的部署目标，网络诱骗系统可以分为产品型的网络诱骗系统和研究型的网络诱骗系统两类。

(1) 产品型的网络诱骗系统一般是一些商业性的网络安全公司以市场为导向开发的面向企业应用的一些专门的网络诱骗系统，其主要部署目标是通过一些虚假的网络资源来耗费攻击者的精力，进而达到保护企业网络安全的目的，如 Resource Technologies 公司的 ManTrap 网络诱骗系统。

(2) 研究型的网络诱骗系统一般是一些信息安全研究人员及相关的研究组织为了对网络的安全状况进行研究而开发的一些网络诱骗系统，其主要部署目标是收集网络上攻击行为的最新动向，为网络安全的研究提供第一手资料，如北京大学计算机所信息安全工程研究中心开发的狩猎女神系统以及 Fred Cohen 开发的 DTK 工具集。

3. 网络诱骗系统部署实施情况分类

按照网络诱骗系统的实施情况，可以将网络诱骗系统分为真实主机网络诱骗系统和虚拟主机网络诱骗系统等两类。

(1) 真实主机网络诱骗系统主要是通过在真实的网络环境中放置一些真实的主机和存在安全漏洞的服务来吸引攻击者的注意。由于这种系统中所有的安全漏洞都是真实服务中

存在的，所以很难被攻击者发现，诱骗的效果比较好。但这种网络诱骗系统中的真实网络环境需要大量的硬件投入，这在一定的程度上增加了系统的成本。

(2) 虚拟主机网络诱骗系统是指在特定的计算机系统中通过软件的方法来模拟计算机网络环境和网络服务的漏洞以达到吸引攻击者注意的目的。这种网络诱骗系统的优点是硬件投入较少，系统的扩展比较灵活。但由于网络环境和网络服务软件的复杂性，全面模拟较为困难，很容易在模拟系统中留下一些痕迹，在一定程度上降低了网络诱骗系统的诱骗效果。

9.2.3 可加载内核模块介绍

可加载内核模块(loodable kemel module，LKM)可以允许系统管理员在一台运行着的 Linux 系统的内核上动态增加或删除功能模块。LKM 的功能非常强大，由于其运行于核心层，其拥有系统的最高权限，可以对系统进行任意更改和操作，因此是学习 Linux 安全编程必须掌握的内容之一。

1. 编写内核模块

下面用一个实例介绍 LKM 编程的基本方法(代码如下)：

```
//file:   try.c
#ifndef __KERNEL__
    #define __KERNEL__
#endif
#ifndef MODULE
    #define MODULE
#endif
#include <linux/module.h>
#include <linux/kernel.h>
static int __init try_init(void)
{
    printk(KERN_EMERG "Init.\n");
    return 0;
}
static void __exit try_exit(void)
{
    printk(KERN_EMERG "Exit.\n");
}
module_init(try_init);
module_exit(try_exit);
```

这个简单的源文件就是一个完整的 LKM，如果读者学习过 Windows 驱动开发，一定会感叹 Linux 开发的简洁。该 LKM 模块的功能简单，就是在加载和卸载时分别向终端输出调试信息“Init”和“Exit”。其中最关键的是两行代码为 module_init()函数和 module_exit()函数，它们分别指定在这个 LKM 加载和卸载时调用的函数。

早期版本的 Linux 使用 gcc 编译该模块，在 2.6 内核中要编程人员自己编写 Makefile 文件，然后使用 make 命令对其进行编译，系统会自动调用 kbuild 来完成内核模块的编译和

链接等工作。

Makefile 的编写方法如下。

```
obj-m:=try.o
KERNELBUILD:=/lib/modules/'uname-r'/build
default:
make-C $(KERNELBUILD) M=$(shell pwd) modules
clean:
rm-rf* .o .* .cmd* .ko* .mod.c .tmp_versions
```

在编写好 Makefile 文件后就可以使用 make 命令对该模块进行编译了，编译出的目标文件为 Hello.ko。

2. 使用内核模块

Linux 提供了一套系统命令用于操作 LKM：

(1) insmod：安装模块；

(2) rmmod：删除模块；

(3) modprobe：比较高级的加载和删除模块，可以解决模块之间的依赖性；

(4) lsmod：列出已经加载的模块及其相关信息；

(5) modinfo：用于查询模块的相关信息，比如作者、版权等。

现在可以尝试使用 insmod 加载 try.ko 了，加载后可以尝试使用 lsmod 列出当前全部的内核模块，使用 rmmod 卸载指定模块，现截取屏幕输出如下。

```
root@tuxus-netlab:/home/tuxus/lkm/try# insmod try.ko
root@tuxus-netlab:/home/tuxus/lkm/try#
Message from syslogd@tuxus-netlab at Wed Apr 30 10:19:49 2008 ...
tuxus-netlab kernel: [ 3307.029760] Init

root@tuxus-netlab:/home/tuxus/lkm/try# lsmod
Module          Size    Used by
try             2688    0
binfmt_misc     12680   1
rfcomm          40856   0
l2cap           25728   5 rfcomm
……………………………………………………………………
processor       31048   1 thermal
fan             5636    0
fbcon           42656   0
tileblit        3584    1 fbcon
font            9216    1 fbcon
bitblit         6912    1 fbcon
softcursor      3200    1 bitblit
vesafb          9220    0
capability      5896    0
```

```
commoncap             8192     1 capability
root@ tuxus- netlab:/home/tuxus/lkm/try# rmmod try
root@ tuxus- netlab:/home/tuxus/lkm/try#
Message from syslogd@ tuxus- netlab at Wed Apr 30 10:20:16 2008 ...
tuxus- netlab kernel:[3333.797833] Exit
```

3．扩展 LKM 功能

Linux 还提供了以下的宏命令用于扩展 LKM 功能。

(1) 输出常用信息

一般的 LKM 模块包含作者、版权及说明等信息，为此，Linux 提供了一组宏命令实现此类功能：

a. MODULE_AUTHOR("author")；

b. MODULE_DESCRIPTION("the description")；

c. MODULE_LICENSE("GPL")；

d. MODULE_SUPPORTED_DEVICE("dev")；

(2) 加载时传递参数

和用户程序一样，LKM 也可以在加载时从控制台获得加载参数，可使用宏命令：MODULE_PARM(var, type)实现该功能，其中 var 是变量名称，type 是变量类型，包括下述种类：

a. b：比特型

b. h：短整型

c. i：整型

d. l：长整型

e. s：字符串型

在传递字符串型的参数时，LKM 中存储参数的变量需要提前声明，然后在加载时由 insmod 赋值，此类变量一般为全局变量。例如：

```
int a=3;
char* str;
MODULE_PARM(a,"i");
MODULE_PARM(st,"s");
```

在通过 insmod 加载该模块时，传递参数格式为：insmod try.ko "a=3", "st=hello world"。

此外，MODULE_PARM()也支持最常用的数组类型。用短线'-'把两个数字分开，分别表示数组参数中的最小位数和最大位数。例如：

```
int array[8];
MODULE_PARM(array,"1-8i");
```

通过命令行代入参数的方法：insmod try.ko"array=38745,123,4000"。

(3) 导出符号

引入一个模块的目的常常是为了对内核功能进行扩展，所以模块一般都会导出符号，以便该符号可以被其他内核模块访问。为此 Linux 为用户提供了宏 EXPORT_SYMBOL(var)用以实现该功能。

9.2.4　Linux 系统调用实现原理

1. CPU 安全保护

Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。程序编写者可以通过系统调用接口在应用程序中调用它们。从某种角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。例如在程序中创建一个新的进程所需调用的 fork 函数就属于系统调用范围。

Intel 的 CPU 代码运行分为 4 个安全级别，而 Linux 只使用了其中的两个，即 Ring0 和 Ring3，一般的用户代码运行于 Ring3 而系统内核代码运行于 Ring0。正常情况下用户代码无法访问 Ring0 的内容，但是在某些情况下，例如进程创建等特殊操作，用户代码需要调用运行于内核的某些特定功能，这就需要通过系统调用进行。

2. 系统调用原理

系统调用的实现原理很简单：系统内部保存一张全局表——sys_call_table，表的每个位置对应某系统调用实现函数的指针，当需要调用某个系统调用函数时，用户层代码首先将该函数需要的参数压栈，将所需系统调用在 sys_call_table 表中的位置存入 eax 寄存器，然后调用一条特殊的指令 int 80h，该指令可以使执行流程从 Ring3 陷入到 Ring0，然后系统内核会从栈内存读出相应的参数和所需函数位置到相应的寄存器中，然后调用相关的内核函数完成用户需求的功能，最后返回执行结果。图 9-1 给出了系统调用的执行过程。

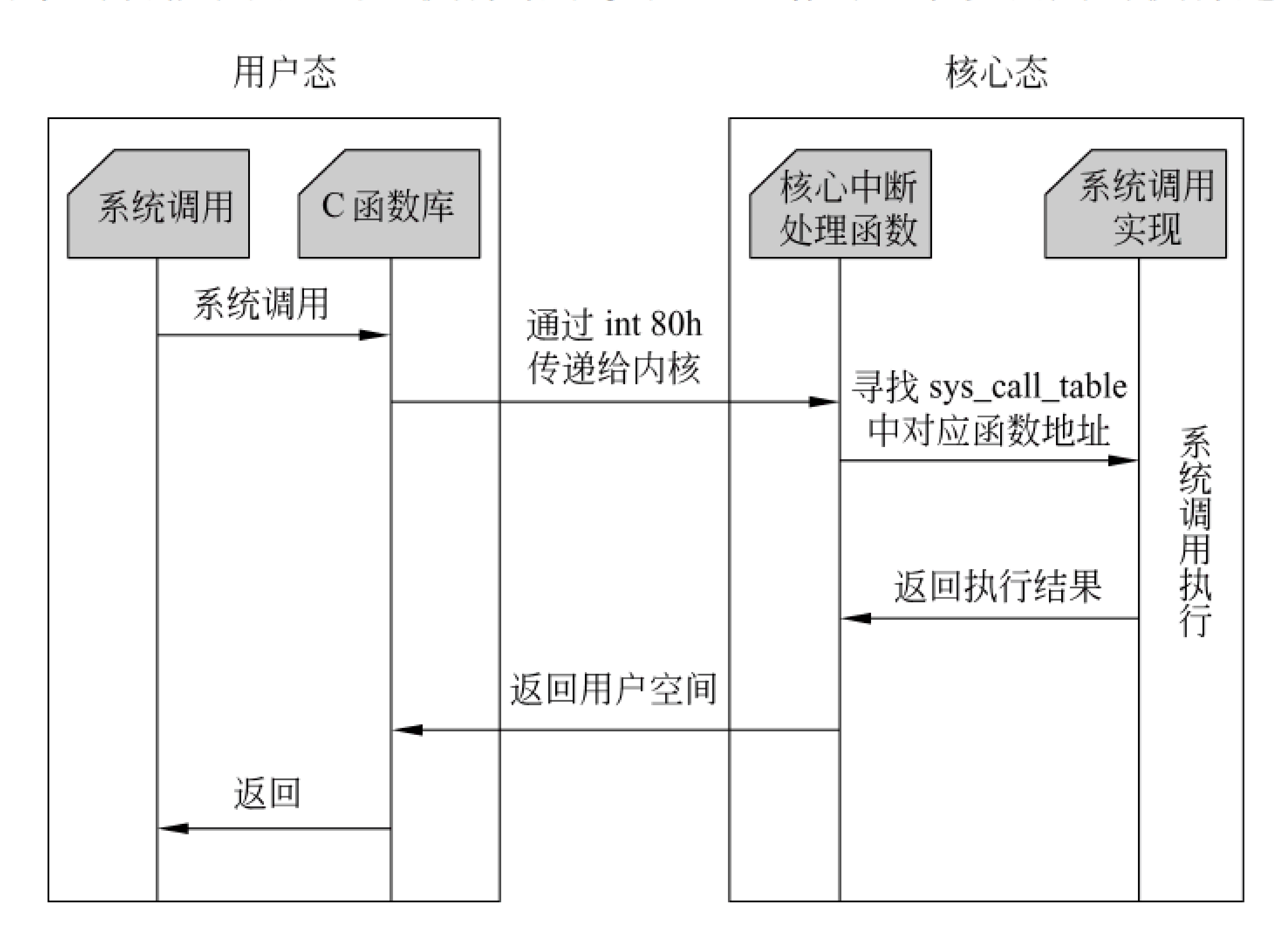


图 9-1　系统调用流程图

3. 系统调用实现

Linux 在 unistd.h 里定义了以下 7 个宏，用于实现参数个数不同的系统调用。

(1) _syscall0(type,name)

(2) _syscall1(type,name,type1,arg1)

(3) _syscall2(type,name,type1,arg1,type2,arg2)

(4) _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)

(5) _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)

(6) _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)

(7) _syscall6(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5,type6, arg6)

其中，type 为返回值的类型，name 为系统调用在 sys_call_table 中的位置，剩下的 argN 和 typeN 即为对应的系统调用参数以及参数类型。

下面的代码为 Linux 实现参数个数为 1 的系统调用代码：

```
#define _syscall1(type,name,type1,arg1)
type name(type1 arg1) {
    __SYS_REG(name)
    register long __r0 __asm__("r0")= (long)arg1;
    register long __res_r0 __asm__("r0");
    long __res;
    __asm__ __volatile__ (
    __syscall(name)
        : "=r" (__res_r0)
        : __SYS_REG_LIST(""(__r0))
        : "memory");
    __res=__res_r0;
    __syscall_return(type,__res);
}
```

可见程序首先将参数存入寄存器中，然后通过调用__syscall(name)宏触发中断，该宏展开之后为 swi #name，即以 name 为中断号触发软中断，然后在中断处理函数中进一步对系统调用进行处理，中断处理函数会将返回值保存在 r0 寄存器中，程序通过调用__syscall_return 将该值返回到用户空间。

```
#define __syscall_return(type, res)
do {
    if ((unsigned long) (res)>= (unsigned long) (-129)) {
            errno=-(res);
            res=-1;
    }
    return (type) (res);
} while (0)
```

为防止和正常的返回值混淆，系统调用并不直接返回错误码，而是将错误码放入一个名为 errno 的全局变量中。如果一个系统调用失败，可以读出 errno 的值来确定问题所在。

9.2.5　Linux 键盘输入实现原理

1. Linux 读取键盘输入的主要流程

(1) 当发生键盘敲击动作时，键盘中断被触发。在中断处理函数 handle_scancode()中，Linux 系统首先产生键盘扫描码，并通过 put_queue()系统调用提交，一个独立的击键行为可以产生长度为 6 的键盘扫描码队列。put_queue()代码如下：

```
static void put_queue(struct vc_data * vc, int ch)
{
    struct tty_struct * tty=vc->vc_tty;
    if (tty) {
            tty_insert_flip_char(tty, ch, 0);
            con_schedule_flip(tty);
    }
}
```

其中，tty_insert_flip_char()函数完成上述队列插入操作。

(2) 然后 Linux 系统根据键盘扫描码将击键操作转换为相应的 key 值，并将其存入 tty_flip_buffer 队列中。

系统通过结构体 tty_ldisc 定义终端接口，每个结构体会定义一组函数指针，用于系统在需要从该终端读或者写时调用。该结构体的定义如下：

```
struct tty_ldisc {
    int             magic;
    char            * name;
    int             num;
    int             flags;
    int             (* open) (struct tty_struct * );
    void            (* close) (struct tty_struct * );
    void            (* flush_buffer) (struct tty_struct * tty);
    ssize_t         (* chars_in_buffer) (struct tty_struct * tty);
    ssize_t         (* read) (struct tty_struct * tty, struct file * file,unsigned char __user * buf,
                     size_t nr);
    ssize_t         (* write) (struct tty_struct * tty, struct file * file,const unsigned char * buf,
                     size_t nr);
    int             (* ioctl) (struct tty_struct * tty, struct file * file,unsigned int cmd, unsigned
                     long arg);
    void            (* set_termios) (struct tty_struct * tty, struct termios * old);
    unsigned int    (* poll) (struct tty_struct * , struct file * ,struct poll_table_struct * );
    int             (* hangup) (struct tty_struct * tty);
    void            (* receive_buf) (struct tty_struct * , const unsigned char * cp,char * fp, int
                     count);
```

```
    void              (* write_wakeup)(struct tty_struct * );
    struct            module * owner;
    int refcount;
};
```

(3) 上层驱动会自动调用 receive_buf()函数从 tty_flip_buffer 中获得字符,然后把这些字符送入 tty read 队列。

(4) 在应用层从键盘读取输入时,应用程序会调用 read()函数从 stdin 中读取数据,read()函数进而调用 sys_read()系统调用,该系统调用根据参数传递的句柄找到相应的 file_operations 结构体,进而调用该结构中注册的 read()函数指针,读取信息。

file_operations 结构体的作用是规定一类操作文件或者设备的接口函数指针。在读取键盘输入时,其中的 read()函数指针调用 tty_read()函数从 tty read 队列中读取数据并将结果返回。

tty_read()函数的实现代码如下:

```
static ssize_t tty_read(struct file* file, char __user* buf, size_t count,loff_t* ppos)
{
    int i;
    struct tty_struct* tty;
    struct inode* inode;
    struct tty_ldisc* ld;
    tty= (struct tty_struct* )file->private_data;
    inode= file->f_dentry->d_inode;
    if (tty_paranoia_check(tty, inode, "tty_read"))
        return- Eio;
    if (!tty|| (test_bit(TTY_io_ERROR, &tty->flags)))
        return- Eio;
    /* We want to wait for the line discipline to sort out in this
    situation* /
    ld= tty_ldisc_ref_wait(tty);
    lock_kernel();
    if (ld->read)
        i= (ld->read) (tty,file,buf,count);
    else
        i= - Eio;
    tty_ldisc_deref(ld);
    unlock_kernel();
    if (i> 0)
    inode->i_atime= current_fs_time(inode->i_sb);
    return i;
}
```

可见,Linux 系统首先获得对应的 tty_ldisc 结构体指针,并使用该指针获得对应设备注册的 read()函数指针进而从该设备中(即上文介绍的 tty read 队列)读取数据。图 9-2 给出了 Linux 读取键盘输入的流程图。

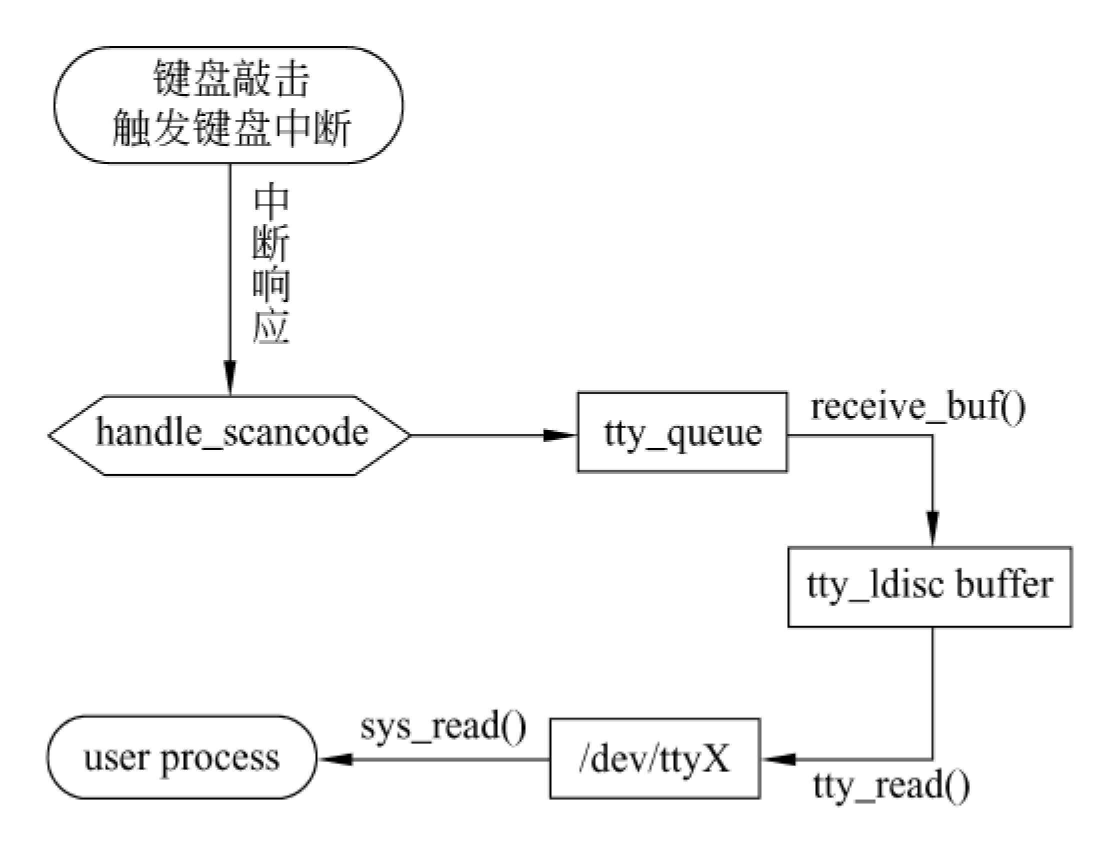


图 9-2　Linux 读取键盘输入流程图

2. Linux 键盘输入截获方式

根据 Linux 键盘操作读取流程,可以使用如下方法截取键盘输入:

(1) 注册新的键盘中断函数。

(2) 劫持 handle_scancode()函数。

(3) 劫持 put_queue()函数。

(4) 劫持 receive_buf()函数。

(5) 劫持 tty_read()函数。

(6) 劫持 sys_read()函数。

在采取劫持函数的方法时,截取越接近键盘驱动层的函数执行效率和隐蔽性越高,但是代码编写也越复杂,考虑到教学需求,本文介绍的示例程序将使用截取 sys_read()函数的方式进行键盘监控,并在扩展提高部分介绍其他的键盘截获方式。

9.3　实例编程练习

9.3.1　编程练习要求

设计一种运行于 Linux 系统的网络诱骗系统,能够监控登录用户在控制台上的输入(键盘操作),并将上述信息以日志的形式记录,要求记录包括输入命令内容,执行者 PID 以及命令输入时间等。

(1) 要求网络诱骗系统程序基于 Linux 系统,使用 LKM 技术进行编写。

(2) 为简化编程难度,推荐读者使用拦截系统调用 sys_read()函数的方式监控用户输入,并假设该程序工作于单核主机上。

(3) 有能力的读者可以尝试实现 LKM 的隐藏功能。

9.3.2　编程训练设计与分析

程序的结构非常简单,分成 3 个模块:系统调用替换模块、输入截获模块和日志记录模

块。整个程序包含在一个 LKM 模块结构中，在模块的初始化函数中，程序完成相关系统调用的替换工作，并初始化全局变量；在卸载函数中，还原系统调用。新的 sys_read()系统调用在任何应用程序执行 I/O 读操作时被调用。在该函数中，程序自动检测 I/O 读取源是否为标准输入(键盘输入)，并记录全部从标准输入读取的内容。

程序执行流程如图 9-3 所示。

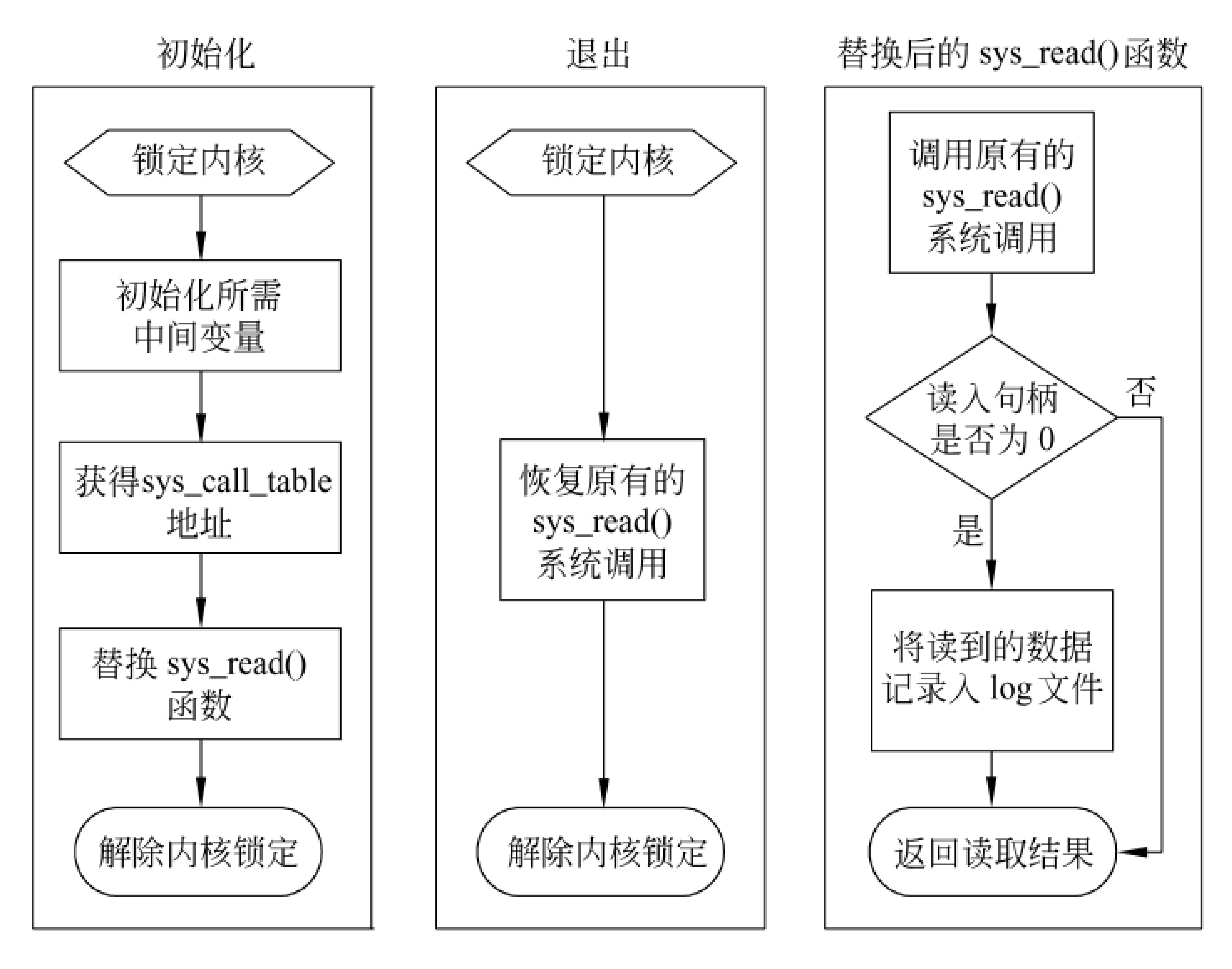


图 9-3 程序执行流程图

1. 获得 sys_call_table 地址

在 Linux 内核 2.6 之前的版本中，系统调用表的地址是通过变量 sys_call_table 导出的，开发人员可以简单的通过访问该变量获得系统调用表的地址，进而对系统进行扩展。但是在 2.6 以后的版本为了安全性考虑不再导出该地址，这就需要程序员手动获得该表的地址。

由于系统调用都是通过 80h 中断来进行的，故在 80h 中断的处理函数中必然能够获得 sys_call_table 的地址。

中断描述符表把中断服务程序和中断向量对应起来，在应用程序进行系统调用时(见图 9-1)，操作系统会调用 80h 中断的处理函数 system_call()。system_call()函数在系统调用表中根据系统调用号找到并调用相应的系统调用服务例程。由于 idtr 寄存器始终指向中断描述符表的起始地址，所以用 sidt 指令得到中断描述符表的起始地址，进而获得 int 0x80 中断描述符所在的位置，然后通过该描述符得出 system_call()函数的地址。

下面的任务就是从 system_call()函数代码中寻找 sys_call_table 的地址。对 system_call()函数进行反编译，可以看到，在 system_call()函数内，是用 call sys_call_table(,eax,4)指令来调用系统调用函数的。因此，只要找到该指令就可以获得系统调用表的地址了。

为了进行查找，必须首先定位查找的特征值，将内核中的 system_call()函数进行反编译，得出如下代码：

```
0xc0103e04 : push %eax
0xc0103e05 : cld
0xc0103e06 : push %es
0xc0103e07 : push %ds
0xc0103e08 : push %eax
0xc0103e09 : push %ebp
0xc0103e0a : push %edi
0xc0103e0b : push %esi
0xc0103e0c : push %edx
0xc0103e0d : push %ecx
0xc0103e0e : push %ebx
0xc0103e0f : mov $0x7b,%edx
0xc0103e14 : movl %edx,%ds
0xc0103e16 : movl %edx,%es
0xc0103e18 : mov $0xfffff000,%ebp
0xc0103e1d : and %esp,%ebp
0xc0103e1f : testl $0x100,0x30(%esp)
0xc0103e27 : je 0xc0103e2d
0xc0103e29 : orl $0x10,0x8(%ebp)
0xc0103e2d : testw $0x1c1,0x8(%ebp)
0xc0103e33 : jne 0xc0103ef8
0xc0103e39 : cmp $0x140,%eax
0xc0103e3e : jae 0xc0103f6b
0xc0103e44 : call * 0xc03094c0(,%eax,4)
0xc0103e4b : mov %eax,0x18(%esp)
```

加黑的一句就是调用 call sys_call_table(,eax,4),而这个 call 指令是本函数中唯一一个 call 指令,所以可以使用它作为搜索的特征值。(call 的指令码为 0xc03094c08514ff)

具体步骤如下。程序首先获取中断描述符表的地址,再从中查找 0x80 中断的服务例程,进而搜索该例程的内存空间,并从其中搜索 sys_call_table 的地址。程序代码如下:

(1) 定义相关的数据结构

```
//中断描述符表寄存器结构
struct {
unsigned short limit;
unsigned int base;
} __attribute__((packed)) idtr;

//中断描述符结构
struct {
unsigned short off1;
unsigned short sel;
unsigned char none, flags;
unsigned short off2;
} __attribute__((packed)) idt;
```

(2) 查找 sys_call_table 地址

```
u32**GetSystemCallTable(void)
{
    unsigned int sys_call_off;
    unsigned int sys_call_table;
    char* p;
    int i;
    asm("sidt %0":"=m"(idtr));
```

上述代码获取 80h 中断处理程序的地址，即 system_call()函数的地址。其原理是：语句 asm("sidt %0":"=m"(idtr))调用汇编指令 sidt 获得中断描述表的地址，并将其保存在结构体全局变量 idtr 中，idtr. base+8 * 0x80 对应的是第 80h 中断的中断描述符的位置，之所以乘以 8 是因为每个中断描述符的大小为 8 个字节(3 个 unsigned short，两个 unsigned char)。

(3) 获得 sys_call_table 地址

然后程序使用 memcpy(&idt, (void *)(idtr. base+8 * 0x80), sizeof(idt))将 int 80h 中断的描述符拷贝到另一个全局变量 idt 中，最后使用 sys_call_off=((idt. off2<<16)| idt. off1)获得 80h 中断处理函数的地址。程序通过 if 语句进行二进制比对的判断寻找 call 语句，找到后获得紧随其后的 sys_call_table 地址，并将其返回；否则返回 0 代表程序执行出错(代码如下)。

```
memcpy(&idt, (void* )(idtr.base+8* 0x80), sizeof(idt));
sys_call_off=((idt.off2<<16)|idt.off1);
p=(char* )sys_call_off;
for (i=0; i<100; i++)
{
    if (p[i]=='\xff' && p[i+1]=='\x14' && p[i+2]=='\x85')
    {
        sys_call_table=* (unsigned int* ) (p+i+3);
        return (u32**)sys_call_table;
    }
}
return 0;
}
```

2. 扩展 sys_read()系统调用

在获得 sys_call_table 的地址后就可以截获原有的 sys_read()系统调用并对其进行扩展了，截获方法非常简单，只需要在 sys_call_table 表中找到相应的位置并将其函数指针更换即可，实现代码如下：

```
lock_kernel();
pOriginalSysCallTable=GetSystemCallTable();
if(NULL!=pOriginalSysCallTable)
{
```

```
        pOldRead= (void * )pOriginalSysCallTable[__NR_read];
        if(NULL!=pOldRead)
        {
            pOriginalSysCallTable[__NR_read]= (u32 * )NewRead;
        }
    }
    unlock_kernel();
```

其中，__NR_read 为 sys_read()函数在 sys_call_table 中的位置。程序将旧的 sys_read()函数地址保存到函数指针变量 pOldRead 中，并使用函数 NewRead()代替之。

NewRead()的实现代码如下：

```
asmlinkage ssize_t NewRead (unsigned int fd, char * buf, size_t count)
{
    char Log[1024];
    ssize_t nRes;
    static int nCmdLength= 0;
    u32 i;
    u32 nUid;
    nRes= pOldRead(fd, buf, count);

    if(nRes< 1 )
    {
        goto OUT;
    }
```

程序调用原有的 sys_read()函数进行读取，如果读取失败或者读取数量为零，则直接返回(代码如下)。

```
if(fd= = 0)
{
    if(buf[((u32)nRes)- 1]= = 13)
    {
        lock_kernel();
        nUid= current-> uid;
        GetTime(DataTime);
        Buffer[nCmdLength]= 0;
        sprintf(Log,
                "< Command Length= %u> < UID= %u> < %s>   %s \r\n",
                nCmdLength,nUid,DataTime,Buffer);
        WriteLog(Log);
        nCmdLength= 0;
        unlock_kernel();
    }
    else
      {
          for(i= 0;i< (u32)nRes;i+ + )
```

```
                {
                    Buffer[nCmdLength]=buf[i];
                    nCmdLength++;
                }
            }
        }
    OUT:
    return nRes
}
```

程序首先调用原有的 sys_read()函数实现读功能,然后判断读取的目标句柄是否为 0,如果为 0,则代表读取的目标设备为标准输入,即键盘终端或者远程终端。如果判断本次读取来自终端,则记录读取到的内容。

由于键盘输入命令一般以回车结尾,且 sys_read()函数在读取终端输入时读取长度不同(普通用户每次读取一个命令,root 用户为了更方便地实时性每次读取一个字符,此外有些网络连接终端受网络条件影响可能每次读取半条命令),记录时首先判断读取的字符串最后一个字符是否为回车,如果是的话,则证明一条完整的命令已经输入,则使用 WriteLog()函数将其记录到日志文件中,否则将读取的字符暂时缓存,直到整条命令接收完毕再统一写入。

其中 current 为一内核导出变量,记录了当前用户的相关信息,代码中使用 current→uid 获得当前登录用户的 UID,GetTime()函数用于获得当前时间和日期,实现细节不再赘述。

在该模块卸载时,其卸载函数会被自动调用,用以恢复被替换的系统调用,实现代码如下:

```
static void __exit HoneyPot_exit(void)
{

    lock_kernel();
    if(NULL!=pOriginalSysCallTable&&NULL!=pOldRead)
    {
        pOriginalSysCallTable[__NR_read]=(u32*)pOldRead;
    }
    unlock_kernel();
}
```

其中 lock_kernel()和 unlock_kernel()为锁定内核函数,其通过申请或者释放大内核锁(BKL-Big Kernel Lock)对内核加以保护。

大内核锁本质上也是自旋锁,但是它又不同于自旋锁,自旋锁是不可以递归获得锁的,因为那样会导致死锁。但大内核锁可以递归获得锁。大内核锁用于保护整个内核,而自旋锁用于保护非常特定的某一共享资源。进程保持大内核锁时可以发生调度,具体实现过程是:在执行 schedule 时,schedule 将检查进程是否拥有大内核锁,如果有,它将被释放,以使其他的进程能够获得该锁,而当轮到该进程运行时,再让它重新获得大内核锁。注意在保持

自旋锁期间是不运行发生进程切换调度的。

需要特别指出，整个内核只有一个大内核锁，这是因为大内核锁是保护整个内核的，而系统只有一个内核，所以只需要一个大内核锁。

3. 记录日志

在内核中，对文件的操作和用户层文件操作基本相同，只不过需要不同的 API 调用。例如：filp_open()函数用于打开文件；filp_close()函数用于关闭相应文件；而对文件进行写操作需要文件对应 file 结构的成员变量 f_op 中的函数指针 write()函数来实现。

在本章代码中为了方便起见，对上述操作进行了封装。封装的代码如下：

```
struct file* klib_fopen(const char* filename, int flags, int mode)
{
    struct file* filp=filp_open(filename, flags, mode);
    return (IS_ERR(filp)) ?NULL : filp;
}

void klib_fclose(struct file* filp)
{
    if (filp)
    {
        filp_close(filp,NULL);
    }
}
```

上述两个函数用于文件的打开和关闭。

```
int klib_fwrite(char* buf, int len, struct file* filp)
{
    int writelen;
    mm_segment_t oldfs;
    if (filp==NULL)
    {
        return-ENOENT;
    }
    if (filp->f_op->write==NULL)
    {
        return-ENOSYS;
    }
    if (((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR))==0)
    {
        return-EACCES;
    }
    oldfs=get_fs();
    set_fs(KERNEL_DS);
```

```
        writelen=filp->f_op->write(filp, buf, len, &filp->f_pos);
        set_fs(oldfs);
        return writelen;
    }
    int klib_fprintf(struct file* filp, const char* fmt, ...)
    {
        static char s_buf[1024];
        va_list args;
        va_start(args, fmt);
        vsprintf(s_buf, fmt, args);
        va_end(args);
        return klib_fputs(s_buf, filp);
    }
```

以上两个函数实现对文件的写操作,其中 klib_fwrite()函数实现了基本的写操作,并在函数内部实现了错误判断等其他扩展功能;klib_fprintf()函数则基于以上函数实现了不定长数据写入文件的功能。

在对文件操作函数进行封装后,日志函数代码的编写就较为简单了。日志写入函数的实现代码如下:

```
    inline void WriteLog(char* pLog)
    {
        struct file* pFile;
        char FileName[256];
        sprintf(FileName,"/tmp/logfile%u.txt",current->uid);
        if ((pFile=klib_fopen(FileName,
                O_CREAT|O_WRONLY| O_APPEND, S_IWOTH|S_IWUSR))==NULL)
        {
            return;
        }
        else
        {
            klib_fprintf(pFile,pLog);
            klib_fclose(pFile);
        }
    }
```

该函数为每个不同的登录用户创建一个独立的日志文件,并在每次调用该函数时用 APPEND 方式打开对应文件,并将新的日志写入文件末尾。

此外,本代码未考虑并发控制,只能在单核系统中正常运行,在多核系统中会造成部分日志丢失,感兴趣的读者可以考虑扩展该函数的功能,通过自旋锁确保该函数关键代码不可重入,从而实现在多核系统中能正常工作。

9.4　扩展与提高

9.4.1　其他键盘输入的截获方法

9.3 节介绍的键盘输入截获方法通过截获系统调用 sys_read()实现，该系统调用是 Linux 系统中调用最频繁的系统调用之一，所以截获该函数会对系统性能造成较大影响。本节介绍通过劫持 receive_buf()函数实现键盘截获的方法。

1. receive_buf()函数

receive_buf()的函数指针在结构体 tty_ldisc 中实现，下面的代码为 tty_struct 的实现代码片段：

```
struct tty_struct {
    int       magic;
    struct tty_driver * driver;
    int index;
    struct tty_ldisc ldisc;
    struct semaphore termios_sem;
    struct termios * termios, * termios_locked;
    char name[64];
    int pgrp;
    int session;
    unsigned long flags;
    int count;
    struct winsize winsize;
    unsigned char stopped:1, hw_stopped:1, flow_stopped:1, packet:1;
    unsigned char low_latency:1, warned:1;
    unsigned char ctrl_status;
    unsigned int receive_room;                  //Bytes free for queue

  //…… 无关代码省略……
}
```

可见，要想实现通过截获 receive_buf()函数截获键盘输入，必须首先访问标准输入设备对应的 tty_struct 结构体，进而访问其成员变量 ldisc 以获得 receive_buf()的函数指针，对该函数进行劫持。

2. 劫持 receive_buf()函数

下列代码实现了简单的劫持功能：

```
struct file * file= fget(fd);
struct tty_struct * tty= file->private_data;
old_receive_buf= tty->ldisc.receive_buf;          //保存原始的 receive_buf()函数
```

```
tty->ldisc.receive_buf=new_receive_buf;          //替换成新的 new_receive_buf 函数
```

程序首先根据待读取句柄 fd 获取其对应的 file 结构体，进而获得其 tty 结构体指针。然后替换该 tty 结构体中 receive_buf()函数指针的值，并保存旧的函数指针。

下列代码实现了新的 receive_buf()函数：

```
void new_receive_buf(struct tty_struct * tty, const unsigned char * cp, char * fp, int count)
{
logging(tty, cp, count);
    (* old_receive_buf)(tty, cp, fp, count);
}
```

其中，logging()函数记录键盘输入的具体内容，在内容记录后，通过调用旧的函数指针实现其原有功能。

在内核中，tty_struct 和 tty_queue 结构仅仅在 tty 设备打开的时候被动态分配。因而，需要通过劫持 sys_ope()系统调用来动态劫持每个 tty 结构体的 receive_buf()函数。新的 sys_open()函数实现代码如下：

```
asmlinkage int new_sys_open(const char * filename, int flags, int mode)
{
    int ret;
    struct file * file;
    ret=(* original_sys_open)(filename, flags, mode);
    if (ret>=0) {
        struct tty_struct * tty;
        BEGIN_KMEM
        lock_kernel();
        file=fget(ret);
        tty=file->private_data;

        if (tty!=NULL &&
            ((tty->driver.type==TTY_DRIVER_TYPE_CONSOLE &&
            TTY_NUMBER(tty)<MAX_TTY_CON-1 ) ||
            (tty->driver.type==TTY_DRIVER_TYPE_PTY &&
            tty->driver.subtype==PTY_TYPE_SLAVE &&
            TTY_NUMBER(tty)<MAX_PTS_CON)) &&
            tty->ldisc.receive_buf!=NULL &&
            tty->ldisc.receive_buf!=new_receive_buf) {
                old_receive_buf=tty->ldisc.receive_buf;
            tty->ldisc.receive_buf=new_receive_buf;
            }
        }
        fput(file);
        unlock_kernel();
      END_KMEM
    }
```

```
    return ret;
}
```

程序首先调用原有的 sys_open()函数打开相应的文件，然后判断该文件句柄是否为标准输入句柄，即键盘输入句柄。如果判断结果为 true，则保存原有的 receive_buf()函数指针并使用自定义的 receive_buf()函数替换原有函数地址，从而实现对系统键盘输入的监控。

截获 sys_open()函数的方法与上节介绍的内容相同。

9.4.2 实现 LKM 在系统启动时自动加载

网络诱骗系统需要在系统运行后自动运行，以确保监控的连续性和可靠性。

Linux 设置启动时自动加载模块非常简单，只需要手动编写脚本，然后使用 chmod＋x 赋予该脚本可执行权限，最后将该脚本拷贝到/etc/init.d 目录下，通过运行命令 chkconfig -add＜脚本名＞可以将该脚本加载为系统启动时自动运行的服务。

此外，编辑/etc/rc.local 配置文件也可以实现在所有用户登录前加载模块，可以通过在该文件里添加新的行来执行相关脚本或者 shell 命令。

Linux 还提供了启动自动加载指定模块的方式：/etc/modules.conf 存储了系统启动时自动加载的模块名称，只需要在其中加入指定的模块名称即可实现该模块的自动加载。也可以使用系统命令 modconf 进行配置。

当然，考虑到上述手段同时为攻击者所了解，攻击者可能在登录后检查上述文件以确保自身的安全，最好使用其他手段隐藏针对上述文件的修改，或者直接修改内核代码实现自动加载网络诱骗系统。

9.4.3 隐藏 LKM 模块

为了防止攻击者通过检查系统加载模块发现网络诱骗系统，必须对网络诱骗系统使用的模块加以隐藏，以增加网络诱骗系统的安全性。

1. Linux 内核模块的存储格式

在 Linux 系统中存储模块信息的结构体定义如下：

```
struct module
{
    enum module_state state;
    //Member of list of modules
    struct list_head list;
    char name[MODULE_NAME_LEN];
    struct module_kobject * mkobj;
    const struct kernel_symbol * syms;
    unsigned int num_syms;
    const unsigned long * crcs;
    const struct kernel_symbol * gpl_syms;
    unsigned int num_gpl_syms;
    const unsigned long * gpl_crcs;
```

```
        unsigned int num_exentries;
        const struct exception_table_entry* extable;
        int (* init) (void);
        void* module_init;
        void* module_core;
        unsigned long init_size, core_size;
        unsigned long init_text_size, core_text_size;
        struct mod_arch_specific arch;
        int unsafe;
        int license_gplok;
#ifdef CONFIG_MODULE_UNLOAD
        struct module_ref ref[NR_CPUS];
        struct list_head modules_which_use_me;
        struct task_struct* waiter;
        void (* exit) (void);
        struct kernel_param refcnt_param;
#endif

#ifdef CONFIG_KALLSYMS
        Elf_Sym* symtab;
        unsigned long num_symtab;
        char* strtab;
        struct module_sections* sect_attrs;
#endif
        void* percpu;
        char* args;
};
```

其中,state 是模块当前的状态。它是一个枚举型变量,可取的值如下:

```
MODULE_STATE_LIVE,MODULE_STATE_COMING,MODULE_STATE_GOING
```

state 可以表示模块当前正常使用中(存活状态),模块当前正在被加载或者是模块当前正在被卸载。

load_module()函数中完成模块的部分创建工作后,把状态置为:MODULE_STATE_COMING。

sys_init_module()函数中完成模块的全部初始化工作后(包括把模块加入全局的模块列表,调用模块本身的初始化函数),把模块状态置为 MODULE_STATE_LIVE,最后使用 rmmod 工具卸载模块时,会调用系统调用 delete_module(),会把模块的状态置为 MODULE_STATE_GOING。这是模块内部维护的一个状态。

成员变量 list 用于访问内核模块列表。所有的内核模块都被维护在一个全局链表中,链表头是一个全局变量 struct module * modules。任何一个新创建的模块,都会被加入到这个链表的头部,通过 modules→next 即可引用到。其中 name 存储模块的名字,该名称一般与模块文件的文件名相同。

宏 THIS_MODULE 用于对当前模块进行访问,其定义是 #define THIS_MODULE

(&__this_module)，__this_module 是一个 struct module 变量，代表当前模块，跟全局变量 current 类似，该变量由头文件 module. h 导出，指向当前内核模块的 module 结构体实体。

查看 module 信息的命令行是 lsmod，此命令行的实现是调用另一个关于 module 的系统调用 sys_query_module()，该函数从 module_list 中顺序取得相应 module 信息，并将相关信息返回。

2. 隐藏方法

基于上述讨论，可以使用如下方法实现模块的隐藏：获得当前模块在内核链表中对应的结构实体位置，并更改其左右两边邻居链表的指针，进而将该实体从链表中摘除，从而达到隐藏的目的。具体实现代码如下：

```
static void hide_module(void)
{
        __this_module.list.prev->next=__this_module.list.next;
        __this_module.list.next->prev=__this_module.list.prev;
        __this_module.list.next=LIST_POISON1;
        __this_module.list.prev=LIST_POISON2;
}
```

上述代码通过更改本模块链表中邻居的链表指针，将自身从本链表中删除，从而实现模块的隐藏。实现原理如图 9-4 所示。

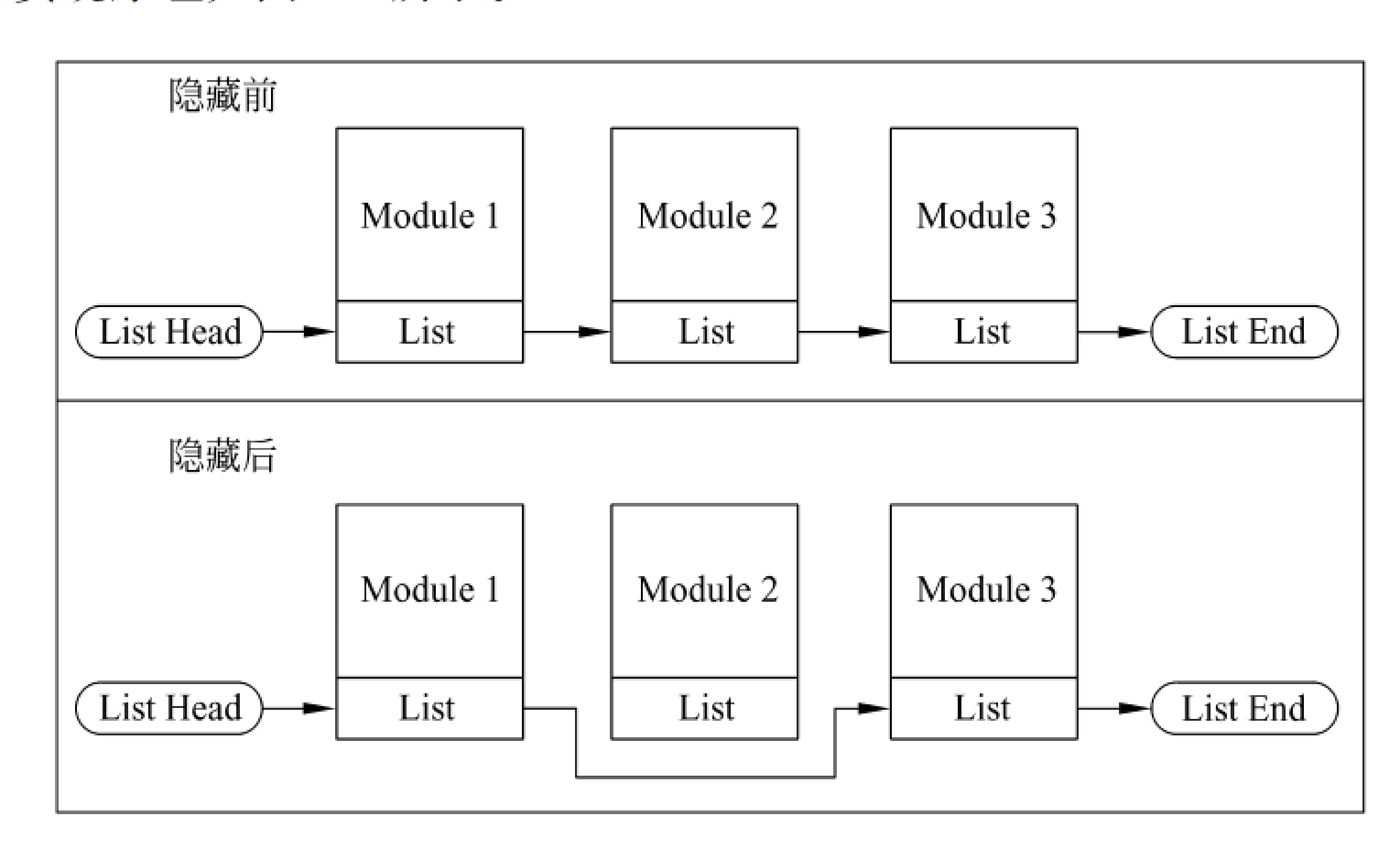


图 9-4　隐藏内核模块原理示意图

3. 内核模块隐藏实验

实验中把该函数添加到内核模块“try”的初始化函数代码中，在编译完成后，发现该模块可以被正常加载，但是其无法通过 lsmod 显示，也无法通过 romod 删除。

截取终端输出如下：

```
root@tuxus-netlab:/home/tuxus/lkm/try# insmod try.ko
root@tuxus-netlab:/home/tuxus/lkm/try#
Message from syslogd@tuxus-netlab at Wed Apr 30 10:21:34 2008 ...
```

```
tuxus-netlab kernel: [ 3411.782529] Init

root@tuxus-netlab:/home/tuxus/lkm/try# lsmod
Module                    Size          Used by
binfmt_misc               12680         1
rfcomm                    40856         0
l2cap                     25728         5 rfcomm
bluetooth                 55908         4 rfcomm,l2cap
ipv6                      268704        12
vmblock                   14628         3
vmmemctl                  9808          0
ppdev                     10116         0
speedstep_lib             6148          0
cpufreq_powersave         2688          0
cpufreq_stats             7360          0
cpufreq_conservative      8200          0
cpufreq_userspace         5408          0
cpufreq_ondemand          9228          0
……………………………………省略部分输出结果……………………………
capability                5896          0
commoncap                 8192          1 capability
root@tuxus-netlab:/home/tuxus/lkm/try# rmmod try
ERROR: Module try does not exist in /proc/modules
root@tuxus-netlab:/home/tuxus/lkm/try#
```

由于在进行该链表元素删除操作时，对应的 LKM 模块已经工作于 Linux 内核的地址空间中，删除的链表元素只是系统中储存的对应该模块的相关信息，故不会影响该模块的正常功能。

9.4.4 隐藏相关文件

为了防止入侵者发现网络诱骗系统的存在，需要对内核文件、日志文件以及启动脚本等相关文件进行隐藏，以确保网络诱骗系统的隐蔽性。

1. Linux 文件系统原理

Linux 支持多种文件系统，为了方便管理，它引入了虚拟文件系统(VFS)对文件系统进行管理，VFS 是物理文件系统与服务之间的一个接口层，它对 Linux 中每个文件系统的所有细节进行抽象，使不同的文件系统在 Linux 核心以及系统中运行的其他进程看来都是相同的。

在 VFS 中每一个文件或是目录对应唯一的一个 inode 节点。文件或是目录的信息存放在 inode 节点中。通过 strace ls 命令可以发现 ls 命令通过系统调用 sys_getdents64()来获取某个目录下的文件和子目录信息，然后将这些信息返回给 is 命令程序进行显示。

sys_getdents64()系统调用的接口如下：

```
intgetdents64(unsigned int fd, struct linux_dirent64 __user * dirent, unsigned int count)
```

其中 fd 是指向所要查询的目录的文件句柄。getdents64()的工作是从 fd 文件句柄所指向的目录中读取所有的文件和目录信息，并将这些信息存入指针 * dirent 所指向的大小为 count 的内存区域，其实现代码如下：

```
asmlinkage long sys_getdents64(unsigned int fd, struct linux_dirent64 __user * dirent, unsigned int count)
{
    struct file * file;
    struct linux_dirent64 __user * lastdirent;
    struct getdents_callback64 buf;
    int error;

    error= - EFAULT;
    if (!access_ok(VERIFY_WRITE, dirent, count))
        goto out;

    error= - EBADF;
    file= fget(fd);
    if (!file)
        goto out;

    buf.current_dir= dirent;
    buf.previous= NULL;
    buf.count= count;
    buf.error= 0;
    error= vfs_readdir(file, filldir64, &buf);
    if (error< 0)
        goto out_putf;
    error= buf.error;
    lastdirent= buf.previous;
    if (lastdirent) {
        typeof(lastdirent->d_off) d_off= file->f_pos;
        __put_user(d_off, &lastdirent->d_off);
        error= count- buf.count;
    }
out_putf:
    fput(file);
out:
    return error;
}
```

通过对这段代码的分析可知，Linux 通过 vfs_readdir()系统调用遍历指定目录的全部文件，并将所得结果通过一个 linux_dirent64 结构体数组返回到用户调用。其中，linux_dirent64 的定义如下：

```
struct linux_dirent64 {
    u64                 d_ino;
```

```
        s64                 d_off;
        unsigned short      d_reclen;
        unsigned char       d_type;
        char                d_name[0];
};
```

其中，d_ino 是该文件或是目录所对应的 inode 号；d_off 表示从这个 linux_dirent64 结构的开始到下一个 linux_dirent64 开始的距离；d_reclen 表示该 linux_dirent64 结构的大小；d_type 代表文件类型；d_name 指向所代表的文件或目录的文件名，其长度不确定。

2. 文件隐藏方式

通过拦截 sys_getdents64()系统调用，更改其返回的 linux_dirent64 数组，从而实现隐藏指定文件的目的，实现代码如下。

本段代码为用于替换原有系统调用 sys_getdents64()的代码，替换方法以及获得原有系统调用函数指针的过程和本章第 3 节介绍的内容相同。

```
long n_getdents64(unsigned int fd, struct linux_dirent64 * dirp,unsigned int count)
{
    struct linux_dirent64 * dir, * ptr,
     * tmp,
     * prev=NULL;
    long i,rec=0,
    ret=(* o_getdents64) (fd, dirp, count);
```

首先通过调用原有系统调用获得程序的结果(代码如下)。

```
if (ret <=0)
    return ret;
if ((tmp=(struct linux_dirent64 * ) kmalloc(ret, GFP_KERNEL))==NULL)
    return ret;
copy_from_user(tmp, dirp, ret);
ptr=dir=tmp;
i=ret;
```

程序在核心层分配空间，并获得原有系统调用返回的内容，由于原有系统调用返回数据拷贝到用户空间，所以必须使用 copy_from_user()系统调用将数据拷贝回内核空间而不是直接使用 mencpy()函数(代码如下)。

```
while (((unsigned long )dir)< (((unsigned long) tmp)+i)) {
    rec=dir->d_reclen;
    if (strncmp("HoneyPot.ko", dir->d_name,strlen("HoneyPot.ko"))==0) {
        if (!prev) {
            ret-=rec;
            ptr=
            (struct linux_dirent64 * ) (((unsigned long) dir)+rec);
```

```
        } else {
            prev->d_reclen+=rec;
            memset(dir, 0, rec);
        }
    } else
        prev=dir;
    dir=(struct linux_dirent64 *)(((unsigned long)dir)+rec);
}
```

上述代码遍历原有 API 返回的结果,并根据文件名匹配进行过滤(本例实现隐藏文件 HoneyPot.ko),最后更改 linux_dirent64 数组的内容(代码如下)。

```
copy_to_user(dirp,ptr,ret);
kfree(tmp);
return ret;
}
```

最后程序将更改后的数据拷贝回用户空间,并返回系统调用结果。

此外,简单的根据文件名判断隐藏文件无法处理文件同名的问题,其扩展性差。文件对应的 inode 数据结构中有数据项 i_uid,用来表示文件所有者的 ID。可以把需要隐藏的文件对应的 i_uid 设置为特殊的值。通过调用系统调用 syslchown()可以改变文件的 i_uid 值。然后在替换后的系统调用 sys_dirents64()中对每个文件或目录对应的 i_uid 进行检查来判断文件是否需要隐藏。

9.4.5 基于 Linux 网络协议栈下层设备驱动实现通信隐藏

为了进一步提高网络诱骗系统的隐蔽性,可以将日志文件服务器和诱骗主机分别设置在两台独立的主机上,并使用网络协议传输日志文件,从而降低攻击者发现的可能,这就要求协议发送过程隐蔽,本节介绍一种直接调用网络协议栈发送数据包的方法,可以躲避 IPtable 等的监测。

使用这种方法首先需要手动构造数据包,这里包括数据包的链路层:包括源物理地址,目的物理地址,协议类型;网络层:一般使用 IP 协议,包括 IP 地址,协议类型等;传输层:考虑到 TCP 协议具有保持连接,协议复杂的特点,一般选取 UDP 协议以及上层数据负载。

构造好数据包后,就可以直接调用 Linux 网络协议栈的下层设备驱动发送数据包了,部分代码如下:

```
//-----lock the output device
spin_lock_bh(&output_dev->xmit_lock);
//------if Interface ready, TX
if(output_dev &&!netif_queue_stopped(output_dev)){
    //------need to synch on pbuf_end decrement
    if(!output_dev->hard_start_xmit(skb, output_dev)){
        //---returnval of 0==success
        s_packets++;
        s_bytes+=skb->len;
```

```
        }
    }else{
        //-----drop the packet
        kfree_skb(skb);
    }
    spin_unlock_bh(&output_dev->xmit_lock);
```

上述代码摘自开源项目 Sebek。

程序首先调用 spin_lock_bh()函数锁定网络设备，然后通过调用 netif_queue_stopped()函数查询当前网络设备的状态，如果网络设备可以使用，则调用 hard_start_xmit()函数直接发送构造好的数据包，否则放弃该数据包的发送。

这里再介绍一个小技巧，在发送数据包的目的主机与源主机在同一局域网内时，可以使用广播地址(FF:FF:FF:FF:FF:FF)作为发送的目的地址，以降低工作难度。

9.4.6 网络诱骗系统的发展趋势

1. Honey Net 欺骗空间技术

欺骗空间技术是通过增加搜索空间来增加入侵者的工作量，从而达到安全防护目的的一种技术。利用计算机系统的多宿主能力(multi-homed capability)，可以在只有一块以太网卡的计算机上部署具有众多 IP 地址的主机，同时保证任意 IP 地址还具有独立的 MAC 地址。

这项技术可用于建立填充一大段地址空间的欺骗网络，且花费极低。现在已有研究机构能将超过 4000 个 IP 地址绑定在一台运行 Linux 的 PC 上。这意味着利用 16 台计算机组成的网络系统，就能做到覆盖整个 B 类地址空间的欺骗。从效果上看，将网络服务放置在所有这些 IP 地址上将毫无疑问地增加了入侵者的工作量，因为入侵者必须在 4 万个以上的 IP 地址上判断其部署的网络服务的真伪。而且，系统模拟的欺骗服务相对更容易被扫描器发现，从而诱使入侵者上当，增加其入侵时间，大量消耗入侵者的资源，使真正的网络服务被探测到的可能性大为减小。

2. 虚拟网络诱骗系统

真实主机网络诱骗系统虽然具有很强的数据采集和控制能力，但系统所需要投入的硬件成本和管理成本十分昂贵，因此世界上各网络诱骗系统的研究组织和相关公司正在积极研究各种虚拟网络诱骗系统。

虚拟网络诱骗系统并非是一种全新的网络诱骗系统，它和传统的网络诱骗系统有着类似的功能，其特点是在单个主机上模拟运行传统网络诱骗系统的各组成部分，以降低网络诱骗系统安装和维护所需要的费用，增加配置和管理的易用性。“虚拟”的含义在于借助于虚拟化软件在单个硬件系统上同时模拟多个彼此独立的诱骗主机，就像传统的网络诱骗系统中各诱骗主机运行在不用的硬件系统上一样。

目前一般将虚拟网络诱骗系统分为自治型虚拟网络诱骗系统和混杂型虚拟网络诱骗系统两类：

（1）自治型虚拟网络诱骗系统

自治型虚拟网络诱骗系统的特点是将整个网络诱骗系统在单独的硬件系统上进行浓缩实现，不仅包括数据捕获部分的功能，还包括数据控制、日志及捕获数据的存放等部分的功能。自治型虚拟网络诱骗系统具有便携性、易插易用性和廉价的系统开销等优点，但同时它也存在以下几个明显的缺点：

① 存在单点故障瓶颈。

② 需要高性能的硬件配置。自治型虚拟网络诱骗系统同样需要实现复杂的功能，这就要求使用大容量的物理内存和高性能的处理器。

③ 安全性不高。由于各虚拟诱骗主机共享硬件系统，这就增加了系统被攻破的可能性，其安全性很大程度上取决于虚拟化软件的安全性。

④ 软件种类受限制。由于很多软件过程十分复杂，利用有限的资源很难对其进行完全地模拟。

（2）混杂型虚拟网络诱骗系统

混杂型虚拟网络诱骗系统是真实主机网络诱骗系统和虚拟化软件结合的产物，这种网络诱骗系统将网络数据包的捕获、数据控制和日志系统都放在一个与诱骗主机所在的硬件系统隔离的系统之上，所有的诱骗主机仍然在一个单独的硬件系统上虚拟运行，这种隔离在保证不大规模增加系统成本的情况下，在很大程度上降低了系统的风险。混杂型虚拟网络诱骗系统主要有以下两方面的优点：

① 安全性高。混杂型虚拟网络诱骗系统对各功能组件进行了分离，在一定程度上降低了系统被攻破的风险。

② 灵活性强。在混杂型虚拟网络诱骗系统中，可以利用多种软硬件来实现数据的捕获和控制，可以根据需要运行不同的诱骗主机。

混杂型虚拟网络诱骗系统虽然通过结合真实主机网络诱骗系统解决了虚拟网络诱骗系统中存在一些缺点，但它也并不完美，主要存在以下两个比较明显的缺点：

① 混杂型虚拟网络诱骗系统需要两台或两台以上的主机，系统的移动不方便，便携性较差。

② 同自治型网络诱骗系统相比，系统的硬件投入有一定程度的增加。

3. 分布式网络诱骗系统

分布式网络诱骗系统技术将网络诱骗系统散布在网络的正常系统和资源中，利用闲置的服务资源来进行欺骗，从而增大了成功欺骗入侵者的可能性。该技术将高交互的虚拟 Honey Net 和低交互的 Honey Pot 作为代理分布在各个网段内。虚拟的 Honey Net 布置在多个局域网内，不仅能够弥补单一 Honey Net 的不足，而且其实际开销也比真实的 Honey Net 要低，并且因为陷阱网络里面有真实的网络服务和攻击者进行交互，所以降低了整个陷阱网络被发现的风险。而网段内布置的低交互的 Honey Pot 能够模拟大量的主机，使得真实主机被攻击的概率大为降低。

4. 其他诱骗技术

(1) 网络流量仿真技术

产生仿真流量的目的是使攻击者通过流量分析不能检测到网络诱骗系统的存在。在网络诱骗系统中产生仿真流量有两种方法。一种方法是采用实时方式或重现方式复制真正的网络流量，这使得网络诱骗系统与真实系统十分相似。第二种方法是从远程产生伪造流量，例如人工模拟部分网络服务行为或者漏洞特征，对入侵者进行欺骗，这种方式具有更优秀的主观能动性。

(2) 网络动态配置技术

真实网络是随时间而改变的，如果网络诱骗系统是静态的，则在入侵者长期监视的情况下会导致欺骗无效。因此，需要动态配置网络诱骗系统所在网络以模拟正常的网络行为，尽量减少其与真实网络环境的差别。此外，诱骗网络应该尽可能地反映出真实系统的特性。例如，如果办公室的计算机在下班之后关机，则欺骗计算机也应该在同一时刻关机。其他的如假期、周末和特殊时刻也必须考虑，从而降低网络诱骗系统被发现的概率。

(3) 多重地址转换

地址的多次转换能将欺骗网络和真实网络分离开来，这样就可以利用真实的计算机替换低可信度的诱骗，增加了间接性和隐蔽性。其基本概念就是重定向代理服务，由代理服务器进行地址转换，使用相同的源和目的地址在诱骗网络中模拟真实系统。一般将重定向后诱骗服务绑定在提供真实服务主机相同类型和配置的主机上，从而提高欺骗的真实性。

(4) 创建组织信息欺骗

如果某个组织提供有关个人和系统信息的访问，那么欺骗也必须以某种方式反映这些信息。例如，如果组织的 DNS 服务器包含了个人系统拥有者及其位置的详细信息，那么就需要在欺骗的 DNS 列表中具有伪造的拥有者及其位置，否则欺骗很容易被发现。而且，伪造的人和位置也需要有伪造的信息，如薪水、预算和个人记录等。

第10章 入侵检测模型的设计与实现

10.1 本章训练目的与要求

入侵检测系统(IDS)是一种对网络传输进行实时监测,并在发现可疑情况时发出警报信息,或采取主动防御措施的网络安全设备。本章在系统分析入侵检测系统基本工作原理的基础上,以基于特征检测的入侵检测系统为对象,研究入侵检测系统的设计与软件编程方法。

本章训练的主要目的是:

(1) 掌握基于特征的入侵检测系统的基本工作原理、设计与实现方法。

(2) 掌握 K-Means 算法的计算过程。

(3) 掌握在网络安全研究中应用数据挖掘技术的基本概念和方法。

本章编程训练的要求如下:

(1) 使用 K-Means 算法对 KDD Cup 1999 数据集进行聚类分析,建立简单的入侵检测模型。

(2) 利用入侵检测模型对测试数据进行预测。

10.2 相关背景知识

10.2.1 KDD Cup 1999 数据集

KDD Cup 1999 数据集是第 3 届知识发现与数据挖掘竞赛所使用的数据集。该数据集来源于一个模拟的美国空军军事网络环境下的局域网流量数据,包含了多种网络攻击与入侵行为。KDD Cup 1999 通过记录上述局域网连续 9 周的原始 TCP dump 数据流量,生成了大约 700 万条连接记录。每一条连接记录反映了某一时刻从源 IP 地址到目的 IP 地址若干个数据包的传输情况。整个数据集分为训练数据集和测试数据集两部分。训练数据集用于提取数据特征,生成数据挖掘模型;测试数据集用于验证模型的效率与正确性。

KDD Cup 1999 数据集对每一条连接记录都进行了分类(label)。所有记录分为正常(normal)与攻击(attack)两大类,其中攻击行为又可进一步分为以下 4 种类型:

(1) DOS:拒绝服务攻击,如 SYN 洪泛攻击。

(2) R2L:来自于远程主机的非法入侵,如猜测密码。

(3) U2R:非法获得本地主机的超级用户权限,如各种“缓冲区溢出”攻击。

（4）Probing：监视与探测，如端口扫描。

除类别(label)以外，KDD Cup 1999 数据集中的每一条连接记录还包含 41 项属性。本章共用到其中的 18 项属性(如表 10-1 所示)。其中，前 3 个属性为符号类型，其他属性为连续类型。

表 10-1 KDD Cup 1999 数据记录的 18 项属性

名　　称	类　　型	说　　明
protocol_type	symbolic	协议类型
Service	symbolic	服务类型
Flag	symbolic	状态标志
src_bytes	continuous	源到目的字节数
dst_bytes	continuous	目的到源字节数
num_failed_logins	continuous	登录失败次数
num_root	continuous	root 用户权限存取次数
Count	continuous	两秒内连接相同主机的数目
srv_count	continuous	两秒内连接相同端口的数目
serror_rate	continuous	“REJ”错误的连接数比例
same_srv_rate	continuous	连接到相同端口数的比例
diff_srv_rate	continuous	连接到不同端口数的比例
dst_host_srv_count	continuous	相同目的地相同端口连接数
dst_host_same_srv_rat	continuous	相同目的地相同端口连接数比例
dst_host_diff_srv_rate	continuous	相同目的地不同端口连接数比例
dst_host_same_src_port_rate	continuous	相同目的地相同源端口连接比例
dst_host_srv_diff_host_rate	continuous	不同主机连接相同端口的比例
dst_host_srv_serror_rate	continuous	连接当前主机有 S0 错误的比例

读者可以登录到 http://archive.ics.uci.edu/ml/databases/kddcup99/kddcup99.html 上下载 KDD Cup 1999 数据集。需要注意的是，KDD Cup 1999 作为一个完备的数据集，包含多个文件。本章编程将使用文件 kddcup.data_10_percent.gz 与 corrected.gz 分别作为入侵检测模型的训练数据集与测试数据集。

10.2.2 K-Means 算法

1. 聚类方法简介

根据具体应用的不同，聚类算法可分为以下 5 类：

（1）划分方法
（2）层次方法
（3）基于密度的方法
（4）基于网格的方法
（5）基于模型的方法

K-Means 算法是一种基于划分方法的聚类分析方法。所谓划分方法是指：在一个由 n 个对象或元组组成的数据库中构建数据的 k 个划分($k<n$)，每个划分表示一个聚类。划分方法将数据划分为 k 个聚类，并且同时要求满足以下两个要求：

（1）每个聚类至少包含一个对象。
（2）每个对象必须属于且只属于一个聚类。

一般来说，判断划分结果优劣的基本原则是：同一个聚类中的对象之间尽可能“接近”，不同聚类中的对象之间尽可能“远离”。通常，聚类算法会选择一个划分标准(称为相似度函数)来判定对象之间的相似度，比如距离等。

2. 聚类分析中的相似度计算方法

聚类分析中的数据类型包括数值属性、二值属性、符号属性、顺序属性和比例属性。因为 K-Means 算法只适用于聚类均值有意义的情况，所以本章只介绍数值属性的相似度计算方法。数值属性所描述对象之间的相似度可以通过计算两个数据对象之间的距离来确定，如欧氏距离、Manhattan 距离和 Minkowski 距离等。

（1）欧氏距离

最常用的距离计算公式就是欧氏距离，定义如下：

$$d(i,j)=\sqrt{(|x_{i1}-x_{j1}|^2+|x_{i2}-x_{j2}|^2+\cdots+|x_{ip}-x_{jp}|^2)}$$

其中对象 $i=(x_{i1},x_{i2},\cdots,x_{ip})$；对象 $j=(x_{j1},x_{j2},\cdots,x_{jp})$。

（2）Manhattan 距离

另一个常用的距离计算方法就是计算 Manhattan 距离，公式定义如下：

$$d(i,j)=|x_{i1}-x_{j1}|+|x_{i2}-x_{j2}|+\cdots+|x_{ip}-x_{jp}|$$

（3）Minkowski 距离

Minkowski 距离是欧式距离和 Manhattan 距离的一个推广，公式定义如下：

$$d(i,j)=(|x_{i1}-x_{j1}|^q+|x_{i2}-x_{j2}|^q+\cdots+|x_{ip}-x_{jp}|^q)^{1/q}$$

其中 q 为一个正整数；当 $q=1$ 时，它代表 Manhattan 距离计算公式；当 $q=2$ 时，它代表欧氏距离计算公式。

若每个变量被赋予一个权值 w，表示其所代表属性的重要性，则带权值的欧氏距离计算公式如下：

$$d(i,j)=\sqrt{(w_1|x_{i1}-x_{j1}|^2+w_2|x_{i2}-x_{j2}|^2+\cdots+w_p|x_{ip}-x_{jp}|^2)}$$

同样，Manhattan 距离和 Minkowski 距离也可以引入权值进行计算。

3. K-Means 算法设计

（1）K-Means 算法的基本思想

K-Means 算法首先随机选取 k 个对象作为初始聚类的中心；然后计算各个数据对象到每个聚类中心的距离，把数据对象划分到离它最近的聚类中心所在的类中；对调整后的新类

重新计算聚类中心，如果相邻两次计算的聚类中心没有任何变化，则说明数据对象的划分结束。本算法的特点是在每次迭代计算中都要考察各个对象的分类是否正确，如果不正确，就需要重新进行分类。在全部数据调整完毕后，再重新计算聚类中心，进入下一次迭代。如果在一次迭代后所有的数据对象都被正确归类，则聚类中心将不会改变。这也标志着算法已经收敛，可以终止聚类过程。

（2）K-Means 算法流程

K-Means 算法的流程用数学语言描述如下：

① 给定大小为 n 的数据集，令聚类次数 $I=1$，选取 k 个数据对象作为初始聚类的中心 $Z_j(I), j=1,2,3\cdots,k$。k 个聚类用 Cluter(j)表示，其中 $j=1,2,3,\cdots,k$。

② 计算每个数据对象与各聚类中心的距离 $D(x_i, Z_j(I)), i=1,2,3\cdots,n, j=1,2,3,\cdots,k$，如果满足 $D(x_i, Z_m(I))=\min\{D(x_i, Z_j(I)), j=1,2,3,\cdots,k\}$，那么 $x_i \in$ Cluter(m)。

③ 按照下式重新计算每个聚类的聚类中心。其中 n_j 为 Cluter(j)中数据对象的数目，$x_i^{(j)}$ 表示 Cluter(j)中第 i 个对象的值。

$$Z_j(I+1) = \frac{1}{n_j}\sum_{i=1}^{n_j} x_i^{(j)}, \quad (j=1,2,\cdots,k)$$

④ 若 $Z_j(I+1) \neq Z_j(I)$，则 $I=I+1$，返回第②步；否则，算法结束。

10.2.3 K-Means 算法的缺点与扩展

1. K-Means 算法的缺点

虽然 K-Means 算法复杂度低，易于实现，但是在处理大规模数据时，聚类效果往往不理想。尤其是在对初始聚类中心数目 k 进行设置时，选取不同的 k 值，聚类结果也大不相同。KDD Cup 1999 数据集除了包含大量的正常数据外，还包含多种攻击数据。虽然这些攻击数据可以划分为 4 大类，但是在使用 K-Means 算法进行聚类时，仅仅将初始聚类中心数目设置成 5(包括正常数据类型的情况)，并不能够获得最佳的聚类效果。因为在 KDD Cup 1999 数据集中，正常数据和攻击数据是多种多样的。以 DOS 攻击为例，它包含了 smurf、neptune、back、teardrop、pod 以及 land 共 6 种攻击方式。不同种类的攻击有着不同的数值特性，即使属于同一类攻击方式，在相似度方面也存在很大差异。有些攻击数据在相似度上与正常数据十分接近，却与同类型的攻击数据相去甚远。因此，在使用 K-Means 算法对 KDD Cup 1999 数据集进行聚类分析时，无论 k 值如何选取，仅仅对数据集进行一次聚类是远远不够的。

2. K-Means 算法的扩展

为了获得更加精确的聚类结果，需要利用 K-Means 算法对 KDD Cup 1999 数据集进行多次聚类。

对一次聚类而言，它的结果包含了多个划分，每一个划分表示一个子类(SubCluster)。每一个子类中可能只包含一种类别的数据，也可能包含多种类别的数据。对于前一种情况而言，说明该子类已经从其他数据中划分出来，代表了唯一的一种类别；而对于后一种情况而言，子类中包含干扰数据，不能用一种类别来对该子类进行标记。在一般情况下，一次聚

类很难划分出若干个完全干净(即不含任何干扰数据)的子类。子类中往往不只含有一种类别的数据。因此判断一个子类是否可以代表一种类别,是否需要再进行一次聚类,需要引入聚类精度的概念。

聚类精度(Cluster Precision)表示一个子类中干扰数据占主类别数据的比例。它的计算过程分为以下 5 个步骤:

(1) 统计子类中不同类别(Label)数据的数目,用 LabelNum[1]、LabelNum[2]…表示。

(2) 在所有类别中找出数据数目最大的一项,将该值记为 MaxLabelNum,并将对应的类别设置为本子类的主类别(MasterLabel)。

(3) 统计所有非主类别的数据的数目,记为 OtherLabelNum。

(4) 计算子类的聚类精度 Cluster Precition=OtherLabelNum/MaxLabelNum。

(5) 判断聚类精度是否符合要求(即是否低于某一精度标准,比如 0.1),若不符合要求,则使用 K-Means 算法对该子类继续进行聚类;否则,用主类别标记该子类。

图 10-1 形象地描述了利用 K-Means 算法进行多次聚类的过程。根节点表示训练数据集。通过 K-Means 算法对其进行一次聚类后,划分出 5 个子类,Cluster0、…、Cluster4。通过计算聚类精度,得出 Cluster1、Cluster2 和 Cluster4 符合精度要求,不再进行任何操作;而 Cluster0 和 Cluster3 不符合要求,需要采用 K-Means 算法继续进行聚类。通过重复上述操作,不符合聚类精度要求的子类不断地被进一步划分,生成了一棵聚类树(Cluster Tree)。树中的每一个叶子节点都代表一个符合聚类精度要求的子类。聚类树的深度则间接反映了采用 K-Means 算法进行聚类的次数。

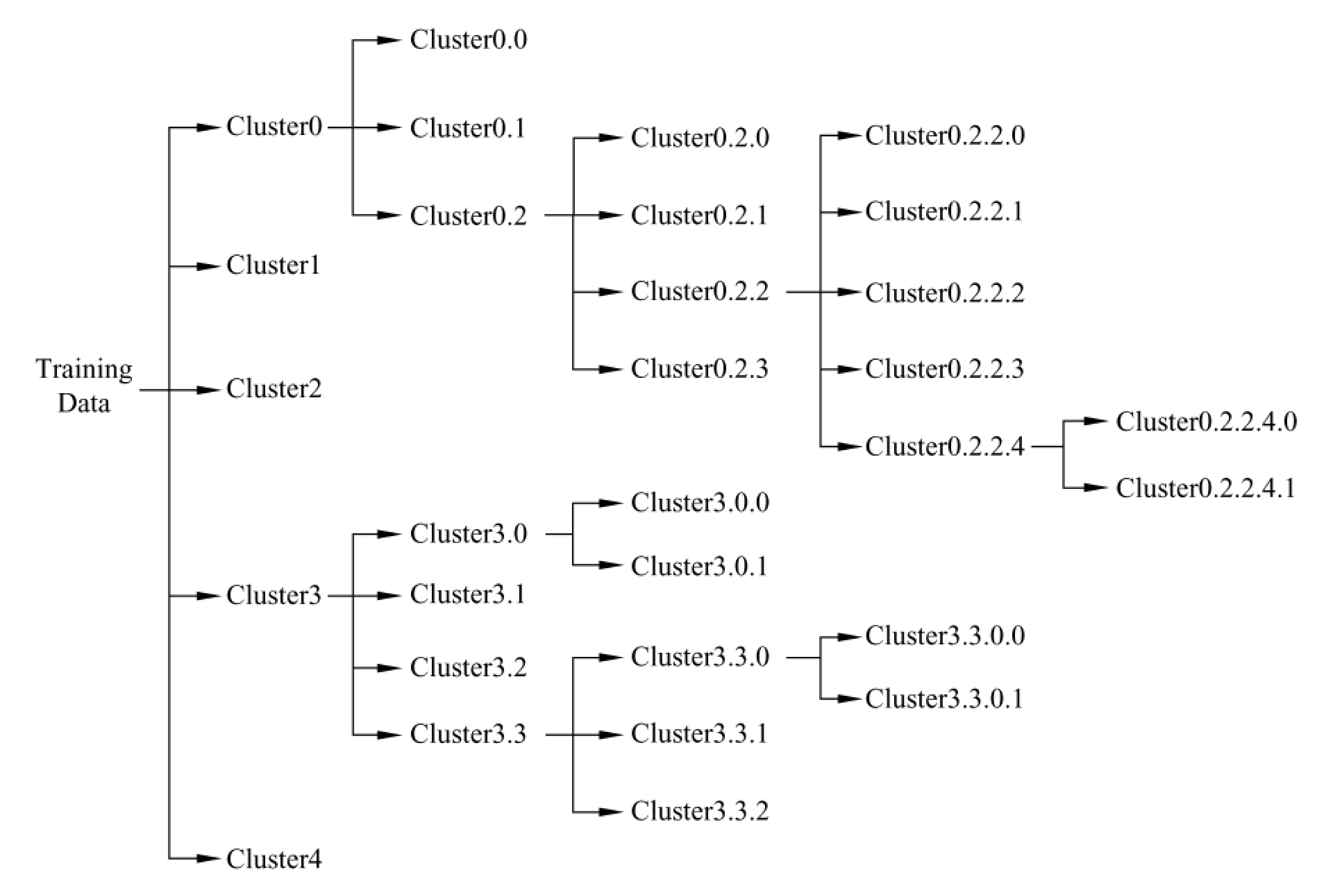


图 10-1 聚类树示意图

此外,采用 K-Means 算法进行多次聚类是一个自动过程。对于每次聚类而言,初始聚

类中心数目 k 的设定也应该是自动的,不需要用户进行手动输入。因此,本章规定在对训练数据集进行首次 K-Means 聚类时,将 k 值设置为 5。这样既包含了正常类别,也包含了 4 种攻击类别。在对子类进行 K-Means 聚类时,k 值设为该子类中所包含的不同类别的数目。采用这种取值方案,是希望在下一次聚类后将子类划分为不同类别的 $k(k\leqslant 5)$个聚类。

10.3 实例编程练习

10.3.1 编程练习要求

本章要求在 Linux 平台上编写 DataPretreat 与 Kmeans 两个应用程序。DataPretreat 程序负责 KDD Cup 1999 数据集的预处理工作。在完成数据集预处理之后,Kmeans 程序使用 K-Means 算法对训练数据集进行多次聚类分析,生成聚类树,建立入侵检测模型,然后读取测试数据集的数据,预测每条记录的类别。下面分别给出编写 DataPretreat 与 Kmeans 程序的具体要求。

1. DataPretreat 程序

DataPretreat 是 Linux 命令行程序,命令行格式如下。

```
./DataPreTreat [输入文件路径]
```

[输入文件路径]指需要进行数据预处理的文件路径。因为无论是对训练数据集进行聚类,还是对测试数据集进行预测,都需要先对数据记录进行预处理,所以 DataPretreat 程序提供了输入不同数据集文件的接口。在进行简单地处理之后,程序将这些数据记录保存到输出文件中。输出文件的命名规则要求在输入文件的名称后面加上"_datatreat"的后缀。如下所示,以 corrected 作为输入文件为例,DataPretreat 程序的执行过程分为两个阶段。首先从输入文件中读取每一条记录,并进行数据处理;然后将处理后的记录写入到输出文件中。因为数据文件比较大,所以无论是在读文件还是在写文件时,都需要及时地输出日志信息。从上面的执行过程中,可以看出 corrected 文件中总共包含了 311029 条记录。所有经过预处理的数据记录都被写入输出文件 corrected_datatreat 中(如下所示)。



```
[root@ localhost DataPretreat]#./DataPreTreat corrected
Start reading records from corrected...
----------10000 lines have read----------
----------20000 lines have read----------
...
All Records have read!Total 311029 records!
Start writing records into corrected_datatreat...
----------10000 lines have written----------
----------20000 lines have written----------
...
All Records have written!
```

2. Kmeans 程序

Kmeans 为 Linux 命令行程序，命令行格式如下：

```
./Kmeans
```

与 DataPretreat 程序相比，Kmeans 程序不需要用户输入任何参数。程序默认将 kddcup. data_10_percent_datatreat 与 corrected_datatreat 分别作为训练数据集与测试数据集文件。这两个输入文件都是由对应文件经过 DataPretreat 程序处理后生成的。Log. txt 与 Result. txt 是 Kmeans 程序的两个输出文件。Log. txt 记录了程序在运行过程中的一些日志信息，比如聚类树的生成过程，每次使用 K-Means 算法进行聚类的结果等等。Result. txt 文件用于保存对测试数据集的预测结果。如下所示，Result. txt 文件的内容包括 3 个部分。Classification Result 部分记录了所有测试数据的预测结果。每一行记录包括测试数据的真实类别(True Label)、预测类别(Pre Label)以及在聚类树中所匹配的聚类路径(Cluster Path)。Total Result 部分对所有测试数据进行统计，包括参加测试的数据总数(Total Test Records)，预测正确的数据数目(Right Label Records)以及预测正确率(Right Rate)。Confusion Matrix 部分显示了预测结果的混淆矩阵。在混淆矩阵中，0～4 分别代表 normal、dos、probe、u2r 和 r2l 5 种不同的类别。矩阵中的 5 列表示预测的分类情况，5 行表示真实的类别。例如，在第 1 行第 2 列的数目是 19，它表示真实类别是 dos 而预测结果是 probe 的测试数据有 19 个。

```
====================Classification Result ====================
True Label=normal Pre Label=normal Cluster Path=0.4.4.4.4.0
True Label=normal Pre Label=normal Cluster Path=0.4.4.4.4.0
...
=======================Total Result=======================
Total Test Records=311029 Right Label Records=299950 Right Rate=0.96438
======================Confusion Matrix======================
T/P    0        1        2      3    4
0      74775    3596     905    0    46
1      143      223136   19     0    0
2      218      120      2039   0    0
3      31       4        0      0    4
4      5442     535      16     0    0
```

10.3.2 编程训练设计与分析

根据编程训练的基本要求，需要分别编写 DataPretreat 与 Kmeans 这两个程序。DataPretreat 程序只负责完成简单的数据预处理工作，而 Kmeans 程序不但需要实现核心的 K-Means 算法，还需要提供聚类树的相关操作，实现 K-Means 算法的扩展功能。

1. DataPretreat 程序的设计与实现

DataPretreat 是提供数据预处理功能的辅助程序。它包括 DataPretreat. h 与

DataPretreat.cpp 等两个文件。如下所示，头文件 DataPretreat.h 中定义了一些基本的数据结构和功能函数。

```
……
#define MAX_BUF_SIZE 512                    //定义字符缓冲区最大值
typedef unsigned char BYTE;                 //定义 BYTE 类型

//**********************全局函数 intToString********************
//将一个 Int 类型的数转换成一个 string
//*******************************************************************
string intToString(int i)
{
    stringstream s;
    s<< i;
    return s.str();
}

//****************数据记录的结构体定义 ************************
struct strMyRecord
{
    BYTE iProtocolType;         //02 协议类型：                 符号变量,3类
    BYTE iService;              //03 服务类型：                 符号变量,66类
    BYTE iStatusFlag;           //04 状态标志：                 符号变量,11类
    int iSrcBytes;              //05 源到目的字节数：           连续变量
    int iDestBytes;             //06 目的到源字节数：           连续变量
    int iFailedLogins;          //11 登录失败次数：             连续变量
    int iNumofRoot;             //16root 用户权限存取次数：     连续变量
    int iCount;                 //232 秒内连接相同主机数目      连续变量
    int iSrvCount;              //242 秒内连接相同端口数目：    连续变量
    BYTE iRerrorRate;           //27 "REJ"错误的连接数比例：    连续变量:0~100
    BYTE iSameSrvRate;          //29 连接到相同端口数比例：     连续变量:0~100
    BYTE iDiffSrvRate;          //30 连接到不同端口数比例：     连续变量:0~100
    int iDstHostSrvCount;       //33 相同目的地相同端口连接数： 连续变量：
    //34 相同目的地相同端口连接数比例:连续变量：0~100
    BYTE iDstHostSameSrvRate;
    //35 相同目的地不同端口连接数比例:连续变量:0~100
    BYTE iDstHostDiffSrvRate;
    //36 相同目的地相同源端口连接比例:连续变量:0~100
    BYTE iDstHostSameSrcPortRate;
    //37 不同主机连接相同端口比例：连续变量:0~100
    BYTE iDstHostSrvDiffHostRate;
    //39 连接当前主机有 S0 错误的比例：连续变量：0~100
    BYTE iDstHostSrvSerrorRate;
    BYTE iLabel;                //42 类型标签：                 符号变量 5类
};
```

结构体 strMyRecord 用于保存从输入文件中读取的数据记录。在 KDD Cup 1999 数据集中，除类别标签（label）以外，每一条记录由 41 项不同的属性值组成。根据 K-Means 算法的要求，结构体 strMyRecord 没有包含所有属性，而是选取了其中的 18 项属性。这些属性分为符号变量与连续变量两种类型。符号变量包括协议类型、服务类型、状态标志以及最后的类别标签。因为 K-Means 算法只适用于均值有意义的情况，所以需要将这些符号变量用数值表示。例如，对协议类型属性而言，就可以用数值 0、1、2 分别代替 icmp、tcp 和 udp 3 种不同的协议。

main 函数是 DataPretreat 程序的核心部分。它的流程如图 10-2 所示。



图 10-2　DataPretreat 程序 main 函数流程图

（1）首先打开输入文件和输出文件。输入文件名 InputFileName 可以通过 main 函数的参数 argv[1]获得。输出文件名则在输入文件名后面拼上后缀字符串“_datatreat”即可。

（2）使用 STL List 模板创建记录链表 RecordList。链表中的每一个节点都是一个 strMyRecord 类型的指针。

（3）调用 fgets 函数读取输入文件中的每一条记录。

（4）对每一条记录进行数据预处理。预处理工作主要包括以下 3 个方面：

① 删除不需要的属性变量；

② 将符号类型变量数值化；

③ 将用百分比表示的变量转换成 0～100 的正整数。

（5）调用 STL List 模板的 push_back 函数，将处理完毕的记录插入记录链表 RecordList 的末端。

（6）遍历记录链表 RecordList，将每条处理过的记录都写入输出文件中。每条记录的不同属性值之间用逗号“，”隔开。

2．Kmeans 程序总体架构

作为编程训练的核心部分，Kmeans 程序负责完成两方面的工作：一方面使用K-Means 算法对训练数据集 kddcup.data_10_percent_datatreat 进行聚类分析，生成聚类树，建立入侵检测模型；另一方面利用入侵检测模型对测试数据集 corrected_datatreat 的数据记录进行类别预测。如表 10-2 所示，Kmeans 程序包含 5 个文件和 3 个类结构。下面将讨论各个类的设计与实现方法。

3．CKMeans 类的设计与实现

CKMeans 类不仅实现了 K-Means 算法的全部计算过程，而且提供了方便的函数接口。一个 CKMeans 类对象表示聚类树的一个节点。它对输入数据进行聚类分析，将不同类别的数据记录插入到不同的子类链表中。如果聚类结果中的某个子类不满足聚类精度的要

表 10-2 Kmeans 程序组成文件列表

文件名	介绍
Common. h	包含一些共用的头文件、宏定义。重新定义了结构体 strMyRecord
Kmeans. h Kmeans. cpp	定义 CKMeans 类。该类实现了 K-Means 算法，提供递归函数 RunKMeans 作为创建聚类树的方法，提供函数 FindClosestCluster 作为预测数据类别的方法
ClusterTree. h ClusterTree. cpp	定义 ClusterNode 类和 ClusterTree 类。ClusterNode 类描述了聚类树的节点信息，每一个节点代表一个聚类。ClusterTree 类提供了聚类树的操作接口，比如插入操作，匹配最近的节点等

求，那么就创建一个新的 CKMeans 类对象对该子类继续进行聚类。总之，聚类树的根节点(Root)将整个训练数据集作为输入，利用 K-Means 算法进行一次聚类后，生成若干个子类作为根节点的子节点。不符合聚类精度的子类会继续使用 K-Means 算法对自身链表中的数据进行聚类分析。这样不断地重复下去，直到聚类树中所有叶子节点都符合聚类精度的要求为止。CKMeans 类的定义如下：

```
class CKMeans
{
public:
    //构造函数 1
    CKMeans(ClusterTree * pTree,int KmeansID,int Level,int NumDimensions)
    //构造函数 2
    CKMeans(ClusterNode * pSelf,ClusterTree * pTree,int KmeansID,int Level,
            int NumDimensions,list< strMyRecord *> * pDataList)
    //读取经过数据预处理的记录
    bool ReadTrainingRecords();
    //K-means 算法的第一步:从 n个数据对象任意选择 k个对象作为初始聚类中心
    void InitClusters(unsigned int NumClusters);
    //K-means 算法的第二步:把每个对象分配给与之距离最近的聚类
    void DistributeSamples();
    //K-means 算法的第三步:重新计算每个聚类的中心
    bool CalcNewClustCenters();
    //计算指定数据对象与聚类中心的欧几里得距离
    float CalcEucNorm(strMyRecord *pRecord, int id);
    //找到离给定数据对象最近的一个聚类
    int FindClosestCluster(strMyRecord *pRecord);
    //打印聚类中心
    void PrintClusters();
    //K-means 算法的总体过程
    void RunKMeans(int Kvalue);
    //在 KMeans 算法运行之后,查询所有聚类的标签数
    void GetClustersLabel();
    //检查聚类后一类中的分类是否合理
    bool IsClusterOK(int i);
    //获得聚类 i 的链表
```

```
        list<strMyRecord*>* GetClusterList(int i);
        //打印本 CKMeans 对象中子类 i 的 label
        void PrintClusterLabel(int i);
        //获得本对象中子类 i 包含不同的 Label 个数
        int GetDiffLabelofCluster(int i);
        //判断是否结束递归过程
        bool IsStopRecursion(int i);
        //创建聚类树节点
        void CreatClusterTreeNode(ClusterNode* pParent);

    private:
        //判断该条记录与之前的聚类中心是否完全相同
        bool IsSameAsCluster(int i,strMyRecord* pRecord);
        //为子类 i 的中心赋值
        void EvaluateCluster(int i,strMyRecord* pRecord);

    private:
        FILE* pInFile;                              //输入文件的指针
        list<strMyRecord*>m_RecordsList;            //数据记录链表
        unsigned int m_iNumClusters;                //聚类的类别数(即 K 值)
        int m_iNumRecords;                          //数据记录的行数
        int m_iNumDimensions;                       //数据记录的维数
        Cluster m_Cluster[MAXCLUSTER];              //子类数组
        int m_ClusterLevel;                         //聚类对象所处的层次
        int m_KmeansID;                             //CKMeans 对象的 ID 号
        ClusterTree* pClusterTree;                  //聚类树的指针
        //本对象孩子节点的指针
        ClusterNode* pClusterNode[MAXLABEL];
        //与本 CKmeans 对象相关的聚类节点的指针
        ClusterNode* pSelfClusterNode;
    };
```

(1) 私有成员变量

文件指针 pInFile 用于读取训练数据集文件，并将数据记录保存到链表 m_RecordsList 中。m_iNumClusters 表示聚类分析中的类别数，即 K-Means 算法中需要预先设定的 K 值大小。m_iNumRecords 和 m_iNumDimensions 分别表示数据记录的行数与维数(即属性的数目)。子类数组 m_Cluster 用于保存聚类结果中每一个子类的信息，比如中心点、成员链表以及成员数目。m_ClusterLevel 表示当前 CKMeans 对象在聚类树中所处的层次。m_KmeansID 表示当前 CKMeans 对象的全局唯一标识符。pClusterTree 是指向聚类树的指针，而指针 pSelfClusterNode 则指向当前 CKMeans 对象在聚类树中对应的节点。指针数组 pClusterNode 用于访问 CKMeans 对象的若干子类在聚类树中所对应的节点。

(2) 构造函数

CKMeans 类有两个构造函数。第一个用于生成首个 CKMeans 对象，作为聚类树的根节点。另一个用于生成聚类树中的子节点。如表 10-3 所示，因为根节点与子节点的初始化

过程不同,所以在参数设置上也略有差异。根节点没有父节点。它的输入数据来源于训练数据集。普通的子节点需要维护自身与父节点的继承关系。它的输入数据是在父节点的聚类结果中不符合聚类精度的子类。

表 10-3 CKMeans 类构造函数参数对比

参　数	说　　明	构造函数 1	构造函数 2
pSelf	指向聚类树中与自身对象对应的节点	无	有
pTree	指向聚类树	有	有
KmeansID	当前 CKMeans 对象的全局唯一标识符	有	有
Level	当前 CKMeans 对象在聚类树中的层次	有	有
NumDimensions	数据的维数	有	有
pDataList	输入数据链表	无	有

(3) 函数 ReadTrainingRecords

函数 ReadTrainingRecords 负责读取 kddcup.data_10_percent_datatreat 文件中的数据,将每一条记录的数据都保存在一个 strMyRecord 结构体中,并用链表 m_RecordsList 将所有的结构体组织起来。只有聚类树的根节点才会调用该函数,而其他节点不需要从训练数据集中读取数据。

(4) 函数 InitClusters

函数 InitClusters 完成 K-Means 算法的第 1 步:从 n 个数据对象中任意选择 k 个对象作为初始聚类中心。函数代码如下:

```
void CKMeans::InitClusters(unsigned int NumClusters)
{
    int i;
    strMyRecord* pRecord;                          //遍历记录的指针
    list<strMyRecord*>::iterator RecdListIter;     //记录链表的迭代器

    m_iNumClusters=NumClusters;                    //对预测的聚类数 k进行赋值
    RecdListIter=m_RecordsList.begin();            //将 List迭代器指向链表头部

    //初始化 m_iNumClusters个类的中心点
    for(i=0;i<m_iNumClusters; i++)
    {
        pRecord=(*RecdListIter);
        //在记录链表中查找出一个与之前的聚类中心数值不同的记录作为中心
        while(IsSameAsCluster(i,pRecord))
        {
            RecdListIter++;
            pRecord=(*RecdListIter);
        }
        //将找到的记录作为一个新的聚类中心保存起来
```

```
            //将记录插入该类的成员链表
            m_Cluster[i].MemberList.push_back(pRecord);
            //将该聚类的成员数设置为 1
            m_Cluster[i].iNumMembers=1;
            //将当前记录的数值赋给第 i 个聚类中心
            EvaluateCluster(i,pRecord);
        }
    }
```

函数首先从链表 m_RecordsList 的头部开始依次读取每一条记录，选择聚类中心。调用函数 IsSameAsCluster 判断当前记录是否与之前选择的聚类中心相同，若是，则放弃当前记录，读取下一条记录；否则，将当前记录作为一个新的聚类中心保存起来。不断重复上面的操作，直到 k 个聚类中心选择完毕为止。

(5) 函数 DistributeSamples

函数 DistributeSamples 完成 K-Means 算法的第 2 步：把每条数据记录划分到与之距离最近的聚类中。具体流程如下：

① 清空所有聚类的成员链表。

② 迭代器 RecdListIter 指向链表 m_RecordsList 的头部。

③ 从 m_RecordsList 链表中读取一条记录。

④ 调用函数 FindClosestCluster，找出与该条记录最近的聚类中心的 ID。

⑤ 将该条记录插入到对应聚类中心的成员链表中。

⑥ 判断记录是否已读取完毕。若是，则退出；否则，迭代器 RecdListIter 指向下一条记录，然后返回第③步继续执行。

在第④步中，函数 FindClosestCluster 调用了函数 CalcEucNorm 分别计算出当前记录到 k 个聚类中心的距离。然后返回距离最近的聚类 ID。函数 CalcEucNorm 的代码如下：

```
float CKMeans::CalcEucNorm(strMyRecord* pRecord,int id)
{
    double fDist;
    fDist=0;

    fDist+=pow((pRecord->fProtocolType-m_Cluster[id].Center[0]),2);
    fDist+=pow((pRecord->fService-m_Cluster[id].Center[1]),2);
    fDist+=pow((pRecord->fStatusFlag-m_Cluster[id].Center[2]),2);
    fDist+=pow((pRecord->fSrcBytes-m_Cluster[id].Center[3]),2);
    fDist+=pow((pRecord->fDestBytes-m_Cluster[id].Center[4]),2);
    fDist+=pow((pRecord->fFailedLogins-m_Cluster[id].Center[5]),2);
    fDist+=pow((pRecord->fNumofRoot-m_Cluster[id].Center[6]),2);
    fDist+=pow((pRecord->fCount-m_Cluster[id].Center[7]),2);
    fDist+=pow((pRecord->fSrvCount-m_Cluster[id].Center[8]),2);
    fDist+=pow((pRecord->fRerrorRate-m_Cluster[id].Center[9]),2);
    fDist+=pow((pRecord->fSameSrvRate-m_Cluster[id].Center[10]),2);
    fDist+=pow((pRecord->fDiffSrvRate-m_Cluster[id].Center[11]),2);
    fDist+=pow((pRecord->fDstHostSrvCount-m_Cluster[id].Center[12]),2);
```

```
        fDist+=pow((pRecord->fDstHostSameSrvRate-m_Cluster[id].Center[13]),2);
        fDist+=pow((pRecord->fDstHostDiffSrvRate-m_Cluster[id].Center[14]),2);
        fDist+=pow((pRecord->fDstHostSameSrcPortRate-m_Cluster[id].Center[15]),2);
        fDist+=pow((pRecord->fDstHostSrvDiffHostRate-m_Cluster[id].Center[16]),2);
        fDist+=pow((pRecord->fDstHostSrvSerrorRate-m_Cluster[id].Center[17]),2);

        return float(fDist);
    }
```

函数将表10-1列出的18项属性作为输入，按照前文给出的欧氏距离的计算方法计算出当前记录与指定聚类中心之间的距离。值得注意的是，函数并没有完全按照前文进行计算。因为欧氏距离的计算结果是否开方并不影响数值大小的比较，所以在函数CalcEucNorm中就省去了开方这一步骤。

（6）函数CalcNewClustCenters

函数CalcNewClustCenters完成K-Means算法的第3步：重新计算每个聚类的中心。函数使用循环分别对k个聚类中心重新进行计算。其中每个聚类中心的计算流程如下：

① 将临时聚类中心TempCenter的每一个属性值都设置为0。

② 遍历当前聚类中心的成员链表，将每一条记录的属性值都加在TempCenter对应的属性上。将TempCenter中的每一个属性值除以聚类成员数，计算出属性的平均值。

③ 判断每个平均值与当前聚类中心的属性值是否不同。只要有一个不同，将新计算出的平均值更新为聚类中心，返回false；否则，聚类中心不变，返回true。

（7）函数RunKMeans

函数RunKMeans将前面介绍的3个函数串联起来，实现K-Means算法的整个过程。函数RunKMeans的流程如图10-3所示：

① 首先调用函数InitClusters，为本次聚类初始化聚类中心。

② 判断bool变量IsFinish的值是否为true。若是，则结束本次聚类，跳转至第⑤步；否则继续下面的步骤。

③ 调用函数DistributeSamples，将每条记录分配给最近的聚类中心。

④ 调用函数CalcNewClustCenters，重新计算聚类中心。如果新的聚类中心与之前的聚类中心相同，则将bool变量IsFinish设为true。

⑤ 调用函数GetClustersLabel，打印本次聚类的结果。包括每个聚类的聚类中心，成员数目以及组成情况。

⑥ 判断对聚类结果的检查是否结束？利用变量i遍历所有的聚类。如果$i \geq$ Kvalue，则所有的聚类检查完毕，跳至第⑩步；否则，继续下面的步骤。

⑦ 调用函数IsClusterOK，判断第i个聚类是否符合聚类精度的要求？若满足要求，跳至上一步；否则，继续下面的步骤。

⑧ 调用函数IsStopRecursion，判断是否需要对第i个聚类继续进行聚类？若是，则继续下面的步骤；否则，跳至第⑥步。

⑨ 创建一个新的CKMeans对象，对第i个聚类的成员继续进行聚类。然后跳转至第⑥步。

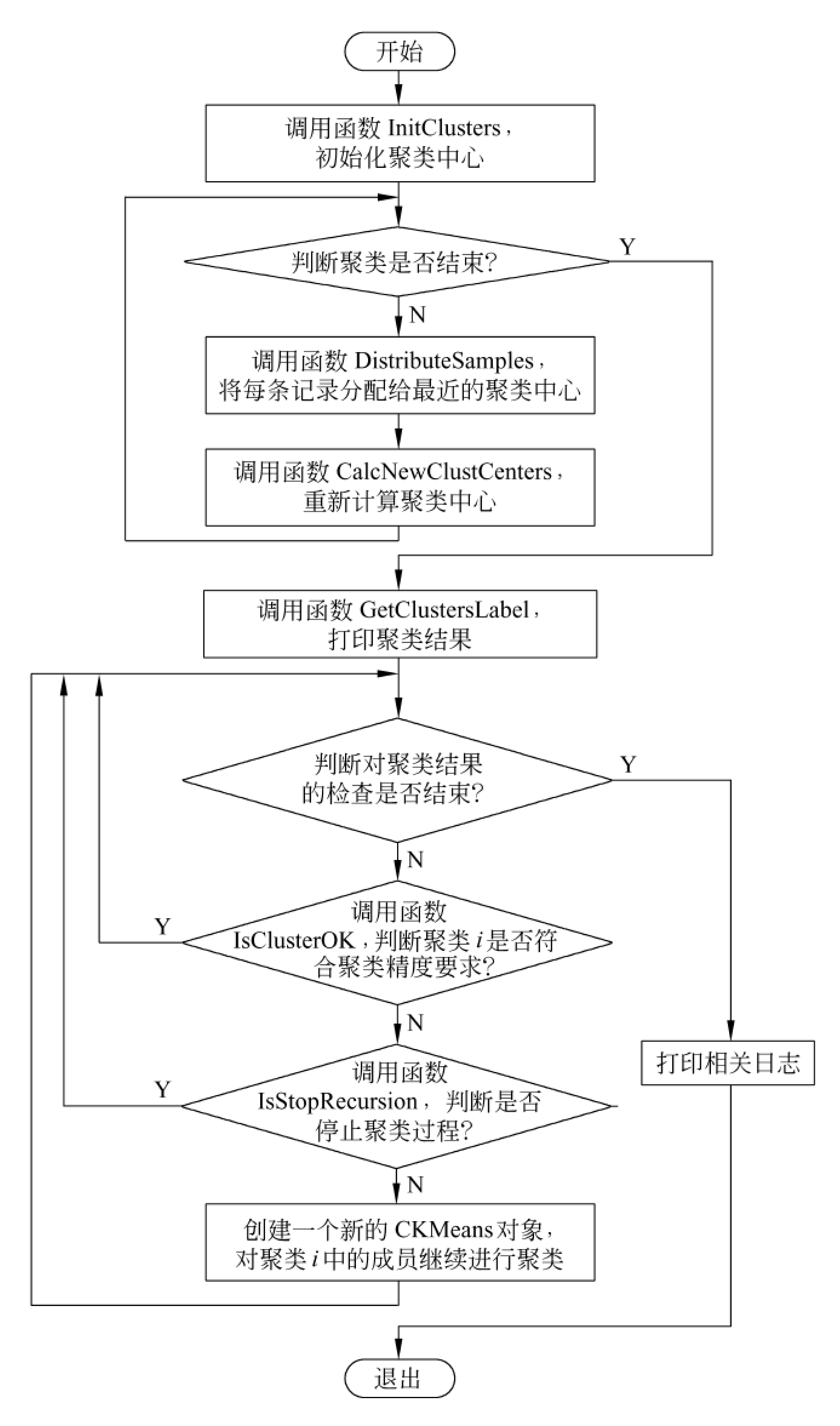


图 10-3　函数 RunKMeans 流程图

⑩ 打印相关日志，退出函数。

4. ClusterNode 类与 ClusterTree 类的设计与实现

(1) ClusterNode 类实现了聚类树中聚类节点的相关操作，为实现 ClusterTree 类提供了底层的操作接口。ClusterNode 类的声明如下：

```
class ClusterNode
{
public:
```

```
        //构造函数
        ClusterNode() {…}
        //计算该条记录到该节点中心的距离
        float CalCenterDistance(strMyRecord * pRecord);
        //获得孩子 i 的指针
        ClusterNode * GetChildNode(int i);
        //获得本节点的聚类标签
        int GetClusterNodeLabel();
        //递归函数,在以该节点为父亲的子树中,获得与数据记录距离最近的聚类节点
        ClusterNode * GetNearestCluster(strMyRecord * pRecord);
        //递归函数,打印节点信息
        void Print();
        //将聚类树输出到日志文件中
        void PrintLog();

    public:
        float fCenter[MAXDIMENSION];        //聚类节点的中心
        string strPath;                     //聚类节点的路径
        ClusterNode * pParentNode;          //指向这个聚类节点的父节点的指针
        ClusterNode *                       //指向这个聚类节点的孩子节点的指针
        pChildNode[MAXLABEL];
        float ClusterResult;                //聚类精度
        bool IsClusterOK;                   //聚合结果是否符合标准
        int IsLeaf;                         //节点类型,是否为叶子节点, 聚类是否正常终止
                                            //0: 非叶子节点
                                            //1: 叶子节点且聚类正常结束
                                            //2: 叶子节点且聚类非正常结束
        int iLabelNum[MAXLABEL];            //记录聚类中各种标签的数目
    };
```

(2) ClusterTree 类将所有聚类节点组织在一棵聚类树中,实现了插入聚类节点、匹配距离最近节点等操作。ClusterTree 类的声明如下:

```
    class ClusterTree
    {
    public:
        //构造函数
        ClusterTree()
        {
            pRootNode= new ClusterNode();
            pRootNode-> strPath= "0";
        }
        //插入节点
        void InsertNode(ClusterNode * pParent,ClusterNode * pNode,int i);
        //找到与给定记录距离最近的聚类节点
        ClusterNode * FindNearestCluster(strMyRecord * pRecord);
```

```
        //获得根节点
        ClusterNode * GetRootNode();
        //打印聚类树
        void Print();
        //将聚类树输出到日志文件中
        void PrintLog();

    private:
        //根节点指针
        ClusterNode * pRootNode;
    };
```

聚类树的生成和使用与 Kmeans 程序的整个运行过程紧密联系在一起。利用 K-Means 算法对训练数据集进行多次聚类分析的过程也是生成聚类树的过程，而对测试数据集中的记录进行预测的过程其实就是在聚类树中为测试记录寻找最匹配的聚类节点的过程。下面分别对这两个过程进行简单介绍。

(3) 聚类树的生成

聚类树的生成过程是 Kmeans 程序对训练数据集进行多次聚类分析的过程，也是建立入侵检测模型的过程。它的主要流程如下：

① 为了对训练数据集进行聚类分析，需要在 Kmeans 程序中创建一个 CKMeans 对象。伴随着这一操作，聚类树完成根节点的插入工作并且将根节点的路径设置为"0"。这些工作都是在 ClusterTree 类的构造函数中实现的。

② CKMeans 对象调用 RunKMeans 函数对训练数据集进行分析。当聚类完成以后，会调用函数 CreatClusterTreeNode 为结果中的每一个聚类创建节点，并将它们作为自己的孩子节点插入到聚类树中。例如，在对训练数据集进行第一次聚类后，生成了 5 个聚类。则就在根节点下插入 5 个子节点，并将这 5 个子节点的路径分别设置为"0.0"、"0.1"、"0.2"、"0.3"、"0.4"。

③ 调用函数 IsClusterOK 进一步判断每一个聚类是否符合聚类精度的要求。如果符合要求，则不需要再进行任何操作；否则，创建一个新的 CKMeans 对象，将该聚类的所有成员作为输入数据，开始新的聚类过程。

由此看出，Kmeans 程序在运行过程中会重复地执行第②步和第③步，不断地向聚类树中插入新的节点。同时也不难看出，在聚类树中，子节点的路径总是在父节点路径的基础上加上新的编号。

(4) 匹配距离最近的聚类节点

在聚类树中为一条数据记录寻找距离最近的聚类节点的过程实际上就是为测试数据预测类别的过程。在编程中采用递归的方法，从聚类树的根节点开始，不断地向下寻找距离最近的节点。ClusterTree 类提供函数 FindNearestCluster 作为对外接口，而 ClusterNode 类的 GetNearestCluster 函数则实现了整个递归过程。上述两个函数的定义如下。

```
    //* * * * * * * * * * * *接口函数,找到与给定记录距离最近的聚类节点* * * * * * * * * * * * *
    ClusterNode * ClusterTree::FindNearestCluster(strMyRecord * pRecord)
    {
```

```
    ClusterNode * pNearestNode;
    //调用根节点的 GetNearestCluster 函数
    pNearestNode= pRootNode- > GetNearestCluster (pRecord) ;
    return pNearestNode;
}
//* * 递归函数,在以该节点为根的子树中,获得与数据记录最近的聚类节点 * *
ClusterNode * ClusterNode::GetNearestCluster (strMyRecord * pRecord)
{
    int i;
    float MinDistance,TmpDistance;                    //最短距离,临时距离
    ClusterNode * pNearestNode;                       //距离最近节点指针
    ClusterNode * pTmpNode;                           //临时节点指针

    //判断是否为叶子节点
    if (IsLeaf> 0)
    {
        pNearestNode= this;
    }
    else
    {
        pNearestNode= this;
        //计算本节点的聚类中心与记录的距离 d1
        MinDistance= CalCenterDistance (pRecord) ;
        //判断是否有孩子节点,如果有,调用递归函数,获得孩子节点所在的子树中
        //离数据距离最近的节点指针
        for (i= 0;i< MAXLABEL;i+ + )
        {
            if (pChildNode[i] != NULL)
            {
                pTmpNode= pChildNode[i]- > GetNearestCluster (pRecord) ;
            }
            //计算最近子节点与数据的距离 d2
            TmpDistance= pTmpNode- > CalCenterDistance (pRecord) ;
            //如果 d2< d1,返回本节点的指针,否则,返回子节点指针
            if (TmpDistance< MinDistance)
            {
                pNearestNode= pTmpNode;
                MinDistance= TmpDistance;
            }
        }
    }
    //返回最近节点的指针
    return pNearestNode;
}
```

在 FindNearestCluster 函数中,仅仅调用了根节点的 GetNearestCluster 函数,在整棵

聚类树中寻找与记录距离最近的节点。而函数 GetNearestCluster 通过自身不断地递归调用完成了对整棵聚类树的查询工作，并返回距离最近节点的指针。当某个节点调用 GetNearestCluster 函数时，它只负责在以自身为父节点的子树中向下寻找距离最近的节点。因此，如果一个父节点需要向下寻找距离最近的节点，首先要让所有孩子节点分别在自身的子树中找出距离最近的节点，然后将这些点的距离与父节点的距离进行比较，从而找出距离最近的节点。函数 GetNearestCluster 就严格地遵循了上述递归思想。如图 10-4 所示，函数 GetNearestCluster 的执行过程分为以下 8 个步骤：

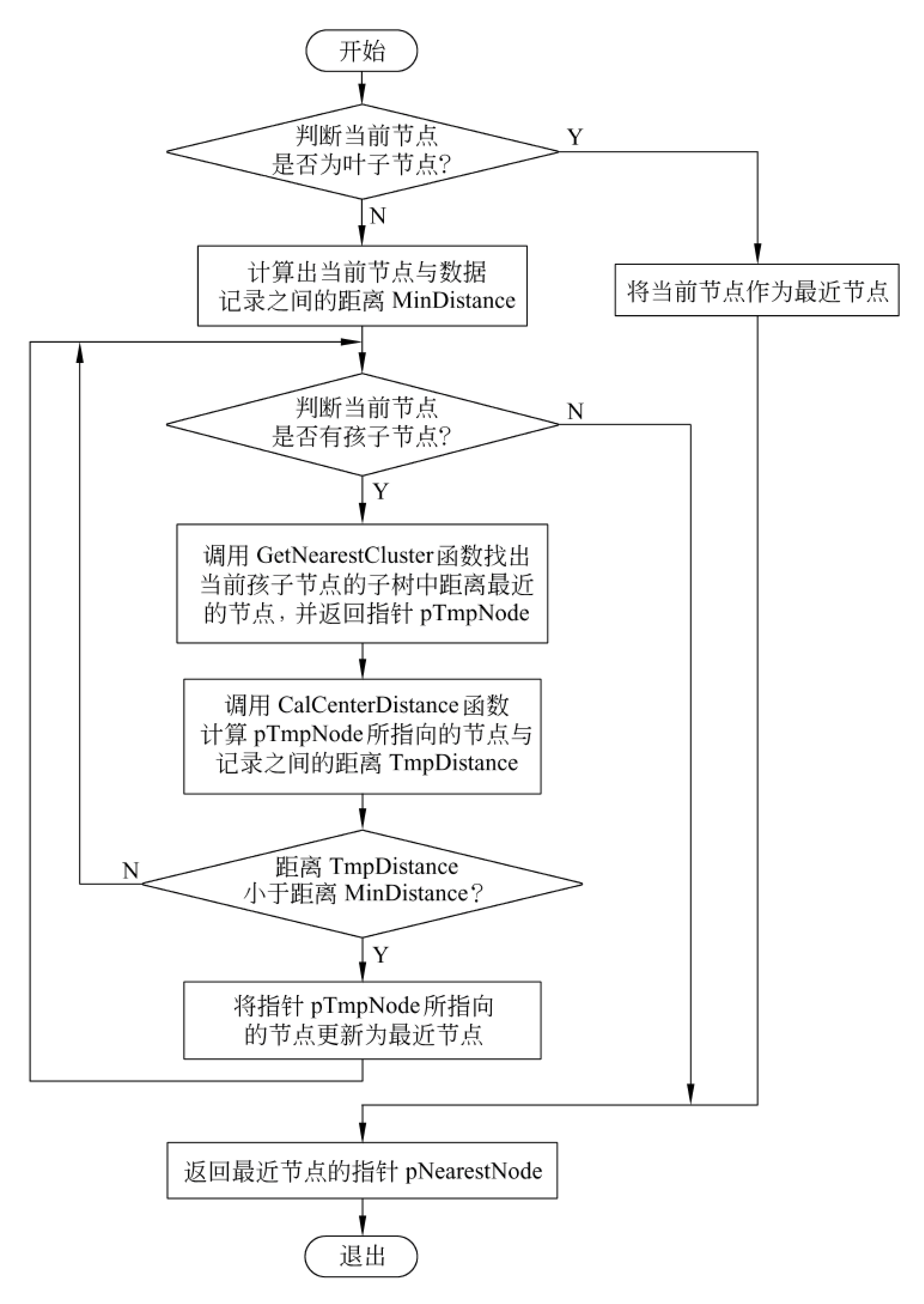


图 10-4　函数 GetNearestCluster 流程图

① 判断当前节点是否为叶子节点。若是，将距离最近节点的指针 pNearestNode 指向当前节点并返回；否则，继续下面的步骤。

② 计算出当前节点与数据记录之间的距离 MinDistance。

③ 判断当前节点是否有孩子节点？若有，继续下面的步骤；否则跳至第⑧步。

④ 调用 GetNearestCluster 函数在当前孩子节点的子树中找出距离最近的节点，并返回指针 pTmpNode。

⑤ 调用 CalCenterDistance 函数计算 pTmpNode 所指向的节点与数据记录之间的距离 TmpDistance。

⑥ 判断距离 TmpDistance 是否小于距离 MinDistance？若是，继续下面的步骤；否则，跳至第③步。

⑦ 将指针 pTmpNode 所指向的节点更新为距离最近的节点。用指针 pNearestNode 指向该节点，并用 MinDistance 表示最近距离的大小。然后跳至第③步。

⑧ 返回最近节点的指针 pNearestNode，退出函数。

5. Kmeans 程序 main 函数的设计与实现

在 Kmeans 程序中，main 函数通过调用 ClusterNode 类与 ClusterTree 类所提供的接口函数将两者有机地结合起来。它不仅使用 K-Means 算法对测试数据集进行多次聚类分析，建立了入侵检测模型，还利用该模型对测试数据集中的数据进行类别预测。如图 10-5 所示，main 函数的执行过程分为以下 8 个步骤：

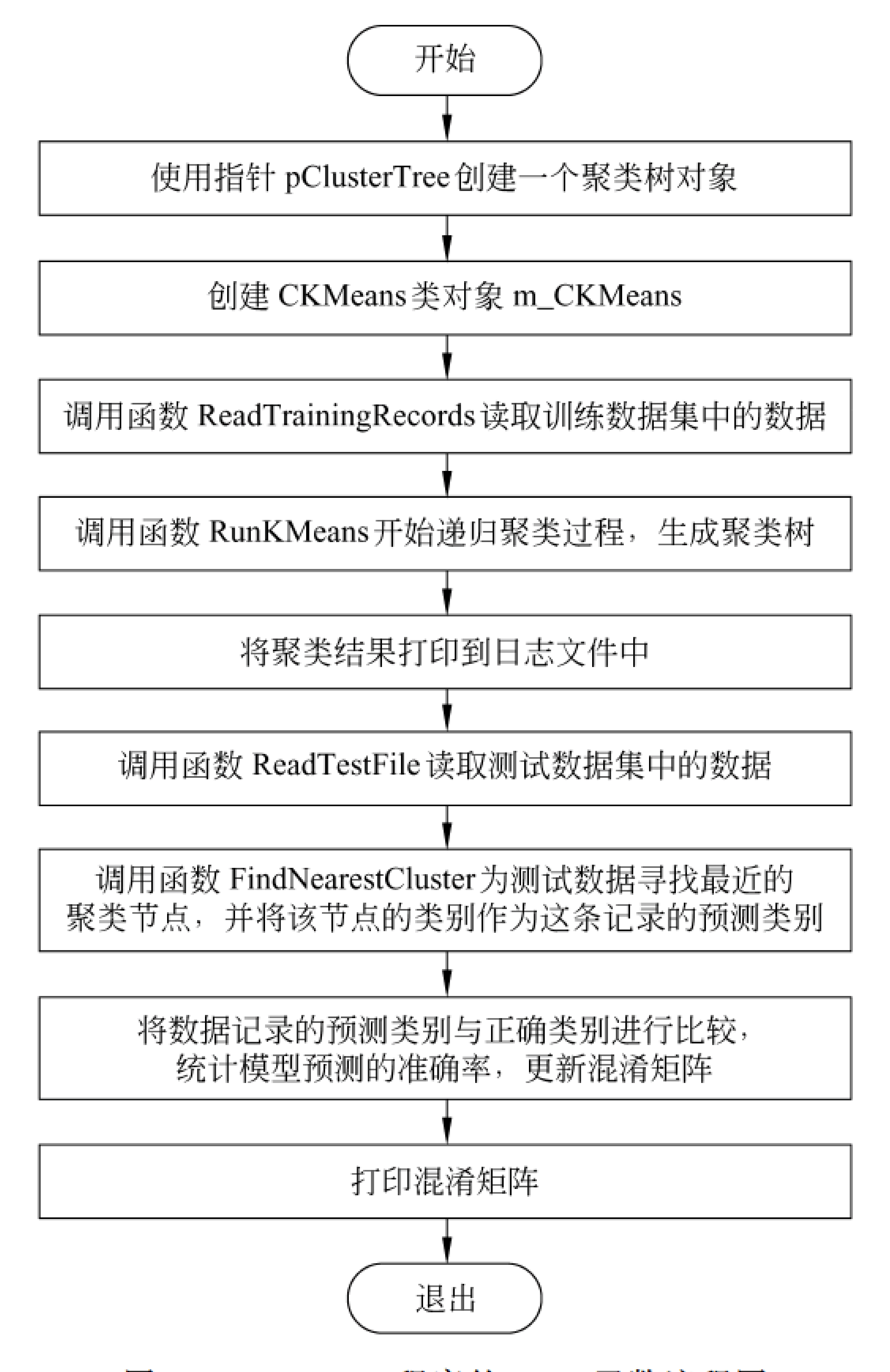


图 10-5　Kmeans 程序的 main 函数流程图

（1）分别创建 CKMeans 类与 ClusterTree 类的对象。用指针 pClusterTree 指向聚类树。将 CKMeans 类对象 m_CKMeans 的 KmeansID 设置为 1。

（2）调用 m_CKMeans 对象的 ReadTrainingRecords 函数，从数据预处理后的训练数据集文件 kddcup.data_10_percent_datatrea 中读取数据。

（3）将 K 值设置成 5，调用 m_CKMeans 对象的 RunKMeans 函数开始递归聚类过程。在此过程中会产生新的聚类节点，需要为这些节点创建新的 CKMeans 类对象，同时将这些节点插入聚类树中。

（4）整个聚类过程结束后，将聚类信息输入到日志文件 Log.txt 中。

（5）调用函数 ReadTestFile 将测试数据集文件 corrected_datatreat 中的测试数据读入链表 pTestRcdList 中。

（6）遍历链表 pTestRcdList。调用函数 FindNearestCluster 为每一条记录寻找聚类树中距离最近的聚类节点。将该聚类节点的类别作为记录的预测类别。

（7）将测试数据的预测类别与真实类别进行比较，将结果记录到 Result.txt 文件中。统计预测结果的准确性，并更新混淆矩阵。

（8）在所有的测试数据预测完毕后，打印混淆矩阵。

10.4　扩展与提高

10.4.1　聚类精度的选取对入侵检测模型的影响

聚类精度(cluster precision)是指一个聚类中干扰数据占主类别数据的比例。聚类精度的大小直接决定了一个聚类是否需要再次使用 K-Means 算法进行聚类分析。但是如何设定聚类精度一直是影响入侵检测模型准确性的重要因素。聚类精度设置过高，会造成类别划分不准确的问题。一个聚类中往往会包含多种类别的数据。这些干扰数据越多，入侵检测模型的准确性就越低。因此，为了提高准确性，通常聚类精度都会选择较小的数值。但是如果聚类精度的数值降低，Kmeans 程序的递归次数将会增加，所占用的资源也会随之增加。有时候，一味追求较小的聚类精度，可能会造成整个程序无法停止，聚类树的生成过程无法收敛的情况。综上所述，合理地选择聚类精度的标准有助于建立高效、准确的入侵检测模型。

然而，只考虑聚类精度是不够的。在聚类精度符合要求的情况下，还会出现一些特殊的问题。例如，假设两个聚类的聚类精度都是 0.1，一个聚类包含 10 条干扰数据和 100 条主类别数据，而另一个聚类包含 1000 条干扰数据和 10000 条主类别数据。显然，前者不需要再继续聚类，而后者虽然达到了聚类精度的标准，但仍包含 1000 条干扰数据。对于这个聚类来说，拥有如此多的干扰数据，完全可以再进行一次聚类分析将这些干扰数据分离开来。鉴于上述考虑，规定聚类中干扰数据的数量可以弥补只根据聚类精度来判定程序是否继续递归的不足，可以有效地防止特殊情况的发生。

根据上述两点，本章的编程中将根据聚类节点在聚类树中所处的不同层次，对聚类精度与干扰数据的数目设定不同的标准。具体方案如下：

（1）首先在每一个 CKmeans 对象中用 m_ClusterLevel 来表示该对象在聚类树中所处的层次。规定根节点的层次为 1，各级子节点的层次依次递增。

(2) CKmeans 类的 IsClusterOK 函数会计算当前的聚类精度并将结果保存在变量 Result 中。然后根据 m_ClusterLevel 变量的不同数值对 Result 进行检查，做出相应的操作。根据聚类层次是否大于 INTERLEVEL(默认设置为 3)，选择聚类精度时分别按照以下两种标准。

① 当 m_ClusterLevel 小于等于 INTERLEVEL 时，首先判断干扰数据的数目是否大于 100，若是，返回 false，要求继续聚类；否则，继续判断聚类精度是否小于 CLUSTER_PRECISION(程序中设为 0.1)，若是，返回 true，不再进行聚类；否则，返回 false。

② 当 m_ClusterLevel 大于 INTERLEVEL 时，首先判断干扰数据的数目是否大于 500，若是，返回 false，要求继续聚类；否则，继续判断聚类精度是否符合要求。利用下式来判断聚类精度是否小于设定的标准，若是，则返回 true，不再进行聚类；否则返回 false。

$$Result < (m_ClusterLevel - INTERLEVEL) \times CLUSTER_PRECISION$$

上述方案对聚类精度的设置如图 10-6 所示。图中 CP 表示当前聚类层次所要求的聚类精度。可以看出，随着聚类层次的不断增大，聚类精度的要求也越来越低。这样既保证了在程序刚开始运行时生成的聚类有较低的精度，也保证了随着聚类层次地不断增加整个聚类过程逐渐达到收敛的状态。

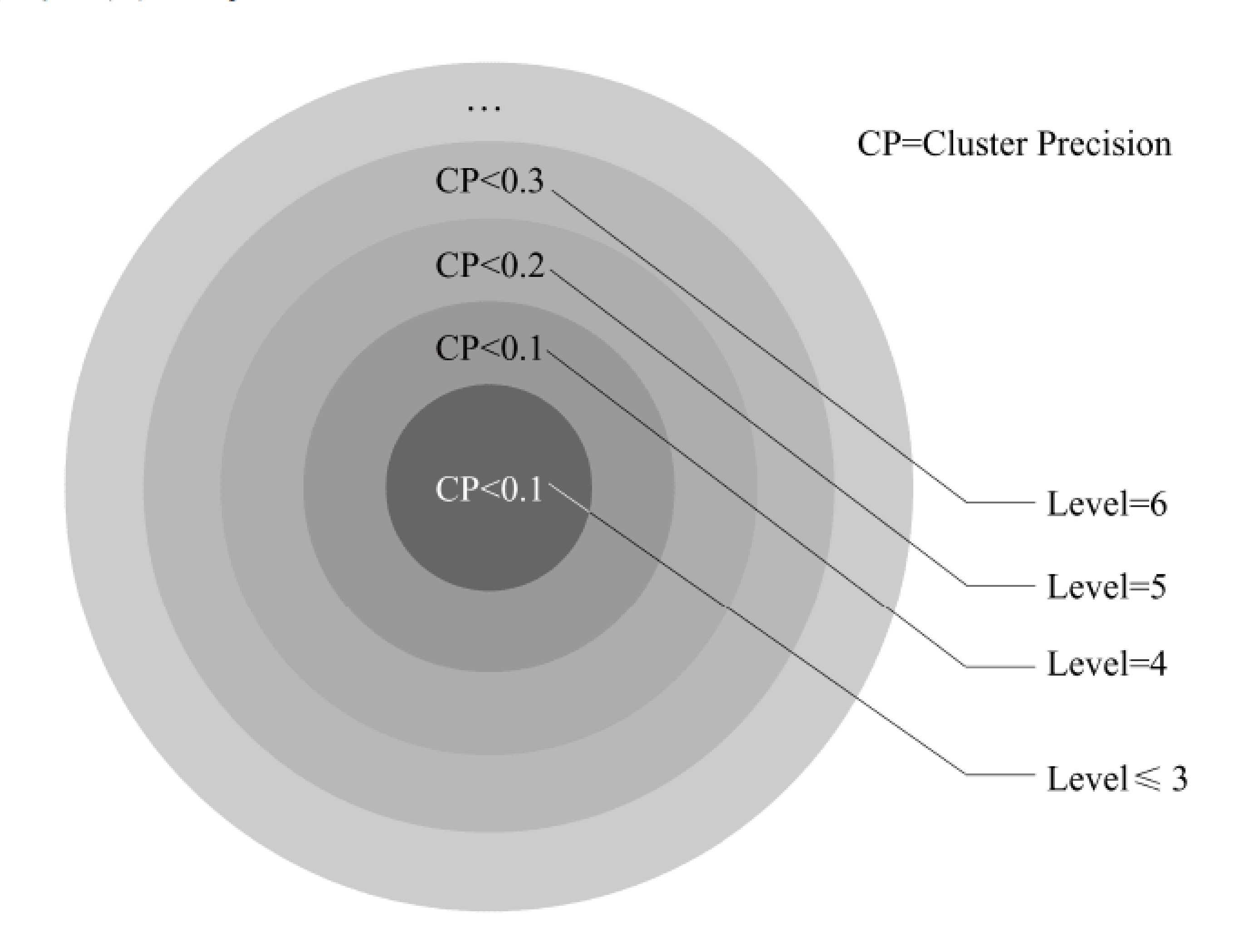


图 10-6 聚类精度设置图

10.4.2 基于 Linux 平台的入侵检测工具

目前，基于 Linux 平台的入侵检测工具有很多，其中最有代表性的是 Snort 和 PSAD。这两款软件具有各自不同的特点。

1. Snort

早在 1998 年，Martin Roesch 编写了一个开放源代码的入侵检测工具，命名为 Snort。迄今为止，Snort 已发展成为一个具有跨平台，能够提供实时流量分析以及记录网络 IP 数据报等特性的入侵检测与防御系统。

Snort 有 3 种工作模式：嗅探器、数据包记录器及网络入侵检测系统。嗅探器模式仅仅是从网络上读取数据包并作为连续不断的数据流显示在终端上。数据包记录器模式把数据包记录到硬盘上。网路入侵检测模式是最复杂的，而且是可配置的。用户可以让 Snort 分析网络数据流以匹配用户定义的一些规则，并根据检测结果采取一定的操作。

网络入侵检测系统(NIDS)是 Snort 最重要的工作模式。可以使用下面的命令行来启动这一模式。

```
./snort- d- h 192.168.1.0/24- l./log- c snort.conf
```

其中 snort. conf 是规则集文件。Snort 会把每个包与规则集进行匹配，发现符合规则的包就采取相应的操作。上述命令是使用 Snort 作为网络入侵检测系统最基本的形式，将符合规则的包记录在日志中，并以 ASCII 形式保存在层次的目录结构里。

Snort 采用规则匹配的方式检测入侵的数据包，因此 Snort 规则的编写十分重要。Snort 使用一种简单的、轻量级的规则描述语言来编写规则。在使用这种语言时，需要掌握以下原则。

Snort 规则分为两个逻辑部分：规则头和规则选项。规则头包含规则的动作、协议、源和目标 IP 地址与网络掩码，以及源和目标端口信息；规则选项部分包含报警消息内容和要检查的包的具体部分。下面是一条规则范例：

```
alert tcp any any→192.168.1.0/24 111(content:"|00 01 86 a5|";msg:"mountd access";)
```

括号前的部分是规则头(rule header)，包含在括号内的部分是规则选项(rule options)。规则选项部分中冒号前的单词称为选项关键字(option keywords)。注意，不是所有的规则都必须包含规则选项部分，选项部分只是为了对符合规则的数据包所进行的操作进行更加严格的定义，比如收集、报警和丢弃等。组成一个规则的所有元素对于指定的操作都必须是真的。当多个元素放在一起时，可以认为它们之间是逻辑与(AND)的关系。同时，Snort 规则库文件中的不同规则之间可以认为是逻辑或(OR)的关系。

2. PSAD

端口扫描攻击探测器 PSAD(The Port Scan Attack Detector)是一种入侵检测工具，它能够检测出不同类型的可疑流量，比如 Nmap 的端口扫描，DDos 攻击以及对系统某个协议的暴力破解等。通过分析防火墙的日志，PSAD 不仅能够检测出某种可疑的攻击，而且还能够通过改变防火墙的规则来响应这一攻击。

PSAD 与 iptables 和 Snort 联系非常紧密。PSAD 在 Linux 系统中运行了 3 个轻量级的系统守护进程。这些守护进程会分析 iptables 的日志信息并检测出端口扫描以及其他可疑的流量。图 10-7 是 PSAD 部署在防火墙 iptables 上的典型结构图。

从图 10-7 中可以看出 PSAD 能够快速地访问防火墙的日志数据。PSAD 还可以利用 Snort 来检测各种后门程序，DDos 工具，以及高级的端口扫描器的探测行为。通过与 Snort 结合，PSAD 能够检测出 Snort 规则集中所描述的各种攻击。此外，PSAD 还能够利用 TCP SYN 数据包的报头字段取得发起扫描行为的远程主机的指纹。

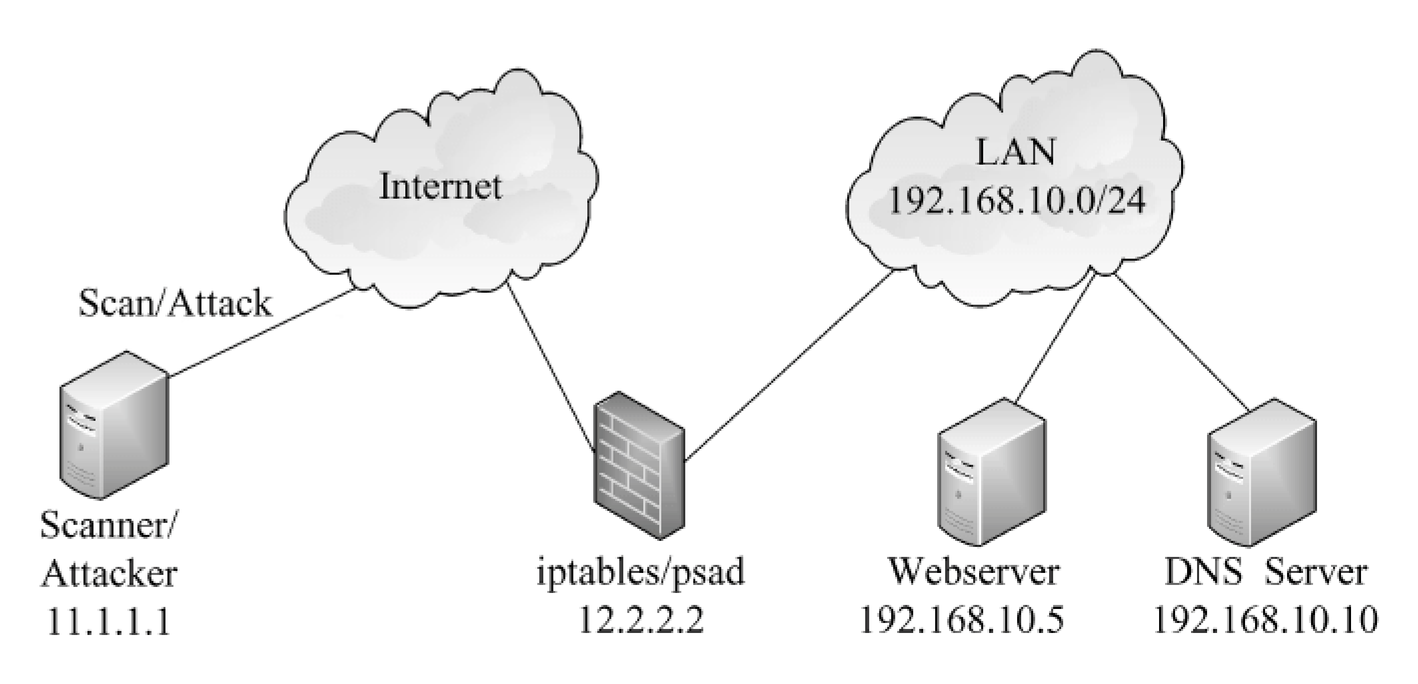


图 10-7 PSAD 部署图

另外,PSAD 还提供了数据输出的接口。用户可以将 PSAD 中的数据导入到软件 AfterGlow 和 Gnuplotd 中。通过这些软件绘制的各种图表进一步分析到底是谁正在攻击防火墙。

第11章 基于Netfilter防火墙的设计与实现

11.1 本章训练目的与要求

网络上充斥着各种病毒、木马以及针对主机漏洞的攻击。如何使网络主机能有效抵御各种非法入侵，保证重要数据的机密性和安全性已成为当前网络上一个亟待解决的问题。防火墙是保护网络资源的重要手段之一。本章以 Netfilter/IPTables 为工具，研究防火墙系统设计与软件编程的方法。

本章训练的主要目的是：

(1) 理解防火墙技术的基本工作原理。

(2) 理解 Linux 环境中 Netfilter/IPTables 的工作机制。

(3) 掌握对 Netfilter 内核模块进行扩展编程的基本方法。

(4) 掌握通过 IPTables 构建防火墙的基本方法。

本章训练要求：

(1) 对 Netfilter 内核模块进行扩展编程来实现简单的防火墙。

(2) 实现基于协议的数据报过滤功能。

(3) 实现基于源 IP 地址的数据报过滤功能。

(4) 实现基于目的端口的 TCP 包过滤功能。

11.2 相关背景知识

11.2.1 防火墙相关知识介绍

1. 防火墙的基本概念

(1) 防火墙的作用

防火墙提供了网络之间或网络对主机的访问控制，以及地址隐藏等技术手段，保护网络资源免受非法侵害，是目前常用的网络安全设备之一。

(2) 防火墙的实现方式

防火墙的具体实现有多种形式，通常是一组硬件设备(路由器和计算机)和适当软件的组合。实现防火墙软件的方式有很多种，有一些应用型防火墙只对特定类型的网络连接提供保护(如针对 SMTP 或 HTTP 协议)。

(3) 防火墙在网络中的位置

防火墙在网络中的位置如图 11-1 所示。客户端对服务器进行访问时不是直接与服务器进行通信的。客户端发送的数据报必须首先经过服务器端防火墙的检测,通过检测的数据报才会转发给服务器,否则将被过滤掉。

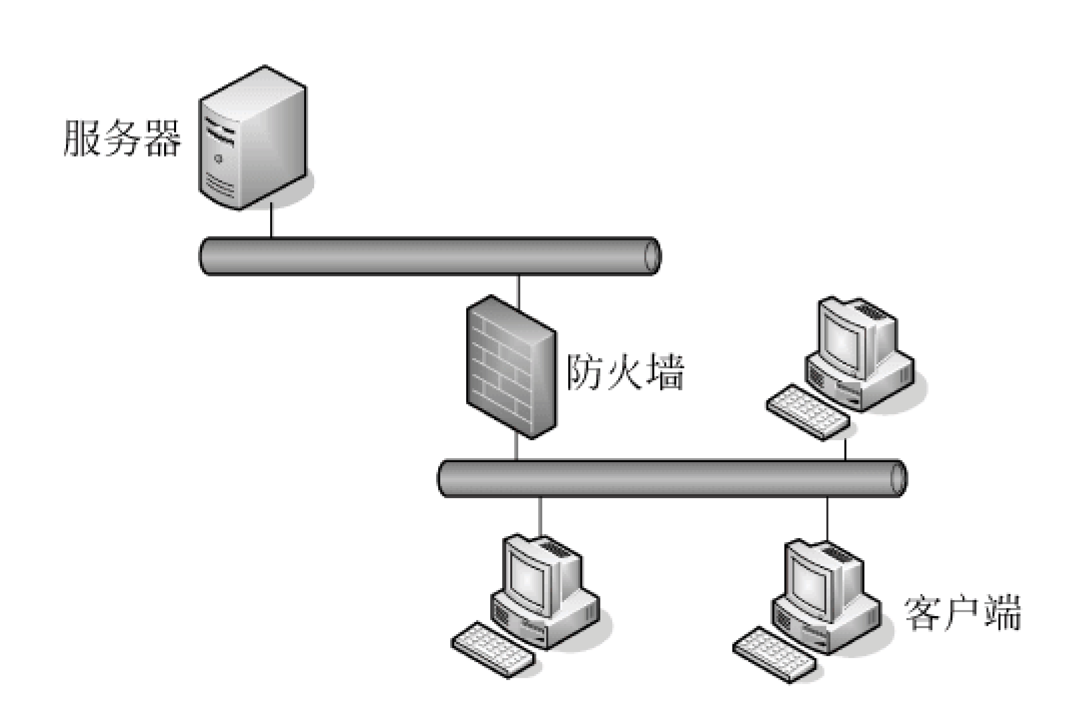


图 11-1　防火墙在网络中的位置示意图

(4) 常见的防火墙类型

① 包过滤型防火墙

包过滤技术作用于网络层(IP 协议),也被称为网络层防火墙。主要是在网络层对数据报进行监控与分析,按照事先设定好的过滤规则,检查数据流中的每个数据报的源地址、目的地址、端口号和协议状态等字段来确定允许哪些数据报通过。其核心主要是过滤规则的制定。

② 应用网关型防火墙

应用网关型防火墙使用代理技术,所以又被称为代理防火墙。它在网络的应用层负责接收外来的应用连接请求,进行安全检查后,再把请求连接到具体的内部网络服务中去。同样内部网络到外部的连接也会被监控。从内部发出的数据报经过这样的防火墙处理后,就好像是从防火墙外部网卡发出一样,从而可以达到隐藏内部网络结构的作用。应用网关型防火墙中的代理技术能完全监视整个连接的过程,并做出详细记录,还能对用户身份进行验证。但和包过滤型防火墙相比,它缺少透明度,对网络性能有一定影响,而且对每一种应用服务都必须有特定的代理模块,实现起来比较困难。

2. Linux 防火墙

Linux 的防火墙技术从最初的设计到现在相对成熟的体系经历了若干代的更替。Linux 2.0 版内核采用一个被称之为 ipforward 的防火墙来操作内核包过滤规则,ipforward 是 Alan Cox 在 Linux 内核发展的初期从 FreeBSD 的内核代码中移植过来的。后来在 Linux 2.2 版内核中 ipchains 取代了 ipforward。由于 ipchains 是在内核级运行的 C 和 C++ 代码,没有很好地提供从用户空间访问 ipchains 的接口,从而导致防火墙应用程序不能使用许多常用的语言编写,这就限制了 ipchains 的可扩展性。后来 Paul Russell 在 Linux 2.3 版内核系列的开发过程中发展了 Netfilter 结构,同时也相应地开发了一个被称之为 IPTables

的管理组件。通过 IPTables 组件，用户可以很方便地在用户空间对 Netfilter 进行设置，从而定制自己的防火墙。本章的训练主要就是通过对 Netfilter 进行内核模块扩展编程来实现防火墙的功能，同时在扩展与提高中讨论如何通过 IPTables 组件来快速搭建一个自定义的防火墙。

11.2.2　Netfilter

1. Netfilter 在内核中的位置

在 Linux 内核中，Netfilter 处在网络层（IP 协议）和防火墙内核功能模块之间，其结构如图 11-2 所示。内核通过 Netfilter 将防火墙对数据报的处理功能在 IP 层中实现，Netfilter 和 IP 层数据报的处理功能可以结合在一起，同时由于它们的结构相对独立，又可以完全分离，构成不同的模块。Netfilter 的设计思想保证了它的灵活性与高效性。Linux 可以支持不同的网络层协议，如 IPv4、IPv6 与 IPX 等。Netfilter 为每种网络协议定义了一套钩子（hook）函数。这些钩子函数在数据报流经协议栈的几个检查点时被调用，它们可以对数据报进行各种处理，如修改、丢弃或传送给用户进程等。

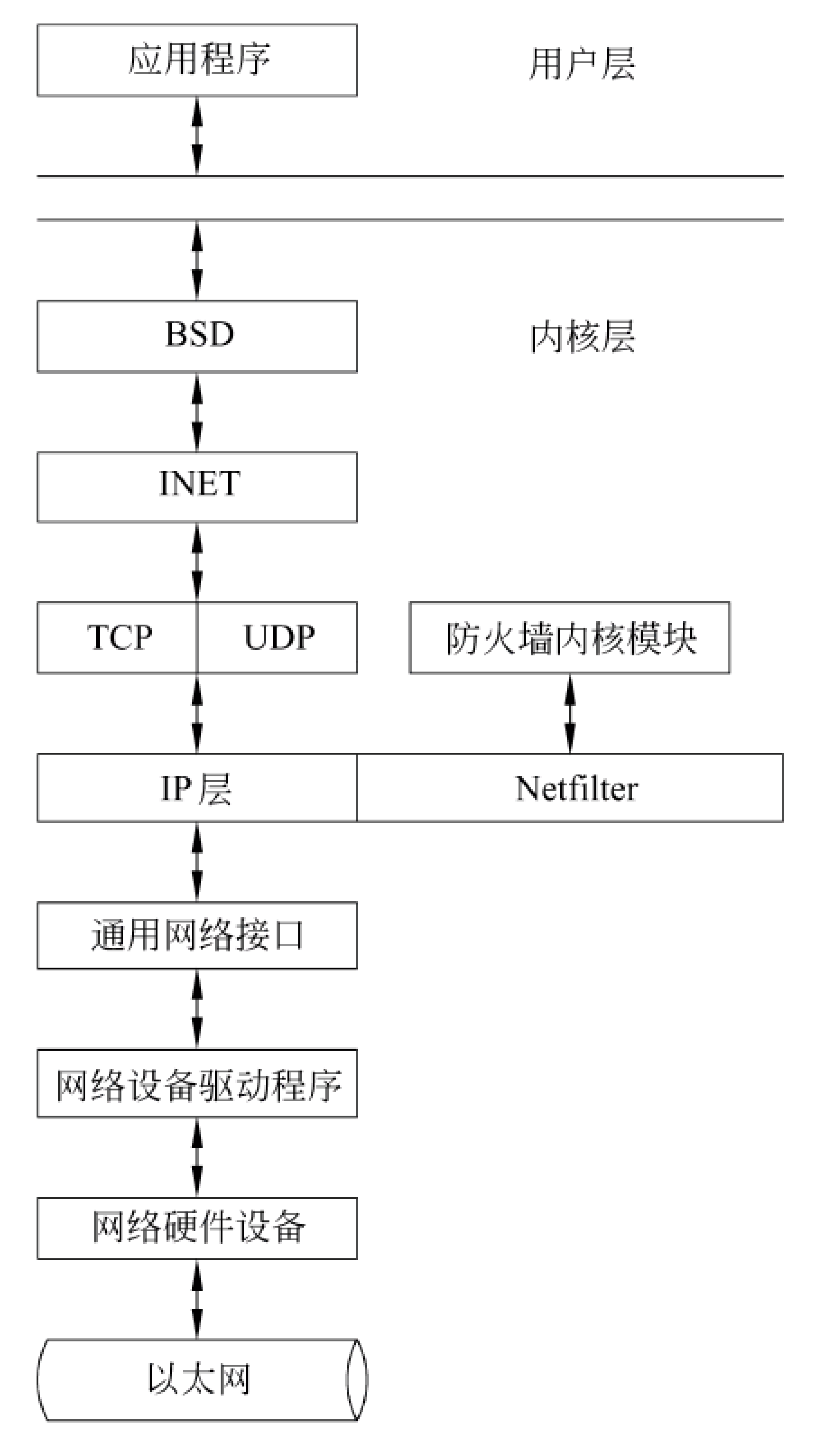


图 11-2　Netfilter 在内核中的位置示意图

2. Netfilter 检查点

网络数据报按照其源 IP 地址和目的 IP 地址，可以分为 3 类：流入的、流经的和流出的数据报。其中流入和流经的数据报需要经过路由才能区分，而流经和流出的数据报则需要通过判断数据包处理流程是否包含从一个 NIC（Network Interface Card）转到另一个 NIC 的转发的过程加以区分。Netfilter 根据网络数据报的流向，在以下 5 个检查点插入钩子函数，这些钩子函数将在数据报流经该检查点时执行。

（1）NF_IP_PRE_ROUTING，在数据报进入路由之前执行。

（2）NF_IP_FORWARD，在数据报转向另一个 NIC 之前执行。

（3）NF_IP_POST_ROUTING，在数据报流出之前执行。

（4）NF_IP_LOCAL_IN，在进入本地的数据报做路由之后执行。

（5）NF_IP_LOCAL_OUT，在本地数据报做流出路由之前执行。

图 11-3 给出了 5 个检查点的位置示意图。

由图 11-3 可知，当某个数据报通过完整性检测以后就会从数据报的入口进入系统，

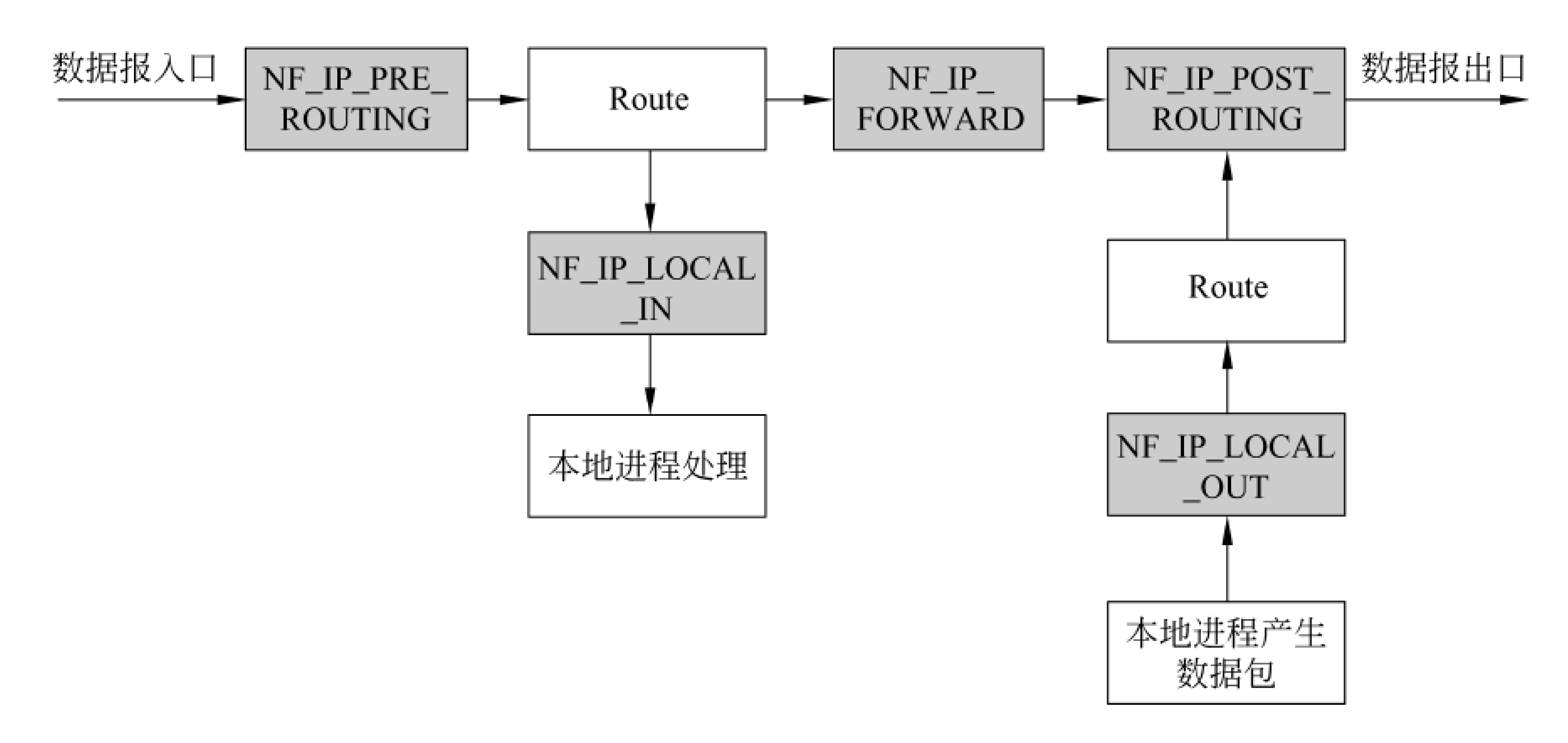


图 11-3 Netfilter 检查点示意图

首先流经 NF_IP_PRE_ROUTING 检查点，接下来由网络协议栈决定数据报是送给本地进程，还是需要转发到其他的网卡，发往本地进程的数据报将流经 NF_IP_LOCAL_IN 检查点；而需要转发给其他网卡的数据报则流经 NF_IP_FORWARD 检查点；在数据报最终发送到网卡驱动程序之前，还要流经 NF_IP_POST_ROUTING 检查点。而对于本地进程发送的数据报，则首先流经 NF_IP_LOCAL_OUT 检查点，然后经过 NF_IP_POST_ROUTING 检查点后进入网络。

3. Netfilter 钩子函数

Netfilter 模块为每种网络协议定义了一套钩子函数，存储在一个 list_head 结构的二维数组中。每个希望嵌入 Netfilter 中的模块都可以在协议族的检查点上注册钩子函数，这些钩子函数将形成一条函数指针链。数据报在协议栈上流经 5 个检查点时会被 Netfilter 模块在这些检查点上注册的钩子函数捕获并分析。Netfilter 模块根据分析的结果，决定数据报的下一步的动作：原封不动地放回 IPv4 协议栈；或者经过一些修改再放回去；或者直接丢弃。

每个注册的钩子函数分析数据报结束后都将返回下列值之一，告知 Netfilter 核心代码分析的结果，以便 Netfilter 模块对数据报采取相应的动作：

（1）NF_ACCEPT：允许数据报通过，进入下一步处理。

（2）NF_DROP：丢弃该数据报。

（3）NF_STOLEN：由钩子函数处理该数据报，不再继续传送。

（4）NF_QUEUE：将数据报加入队列，交由用户程序处理。

（5）NF_REPEAT：再次调用该钩子函数。

11.2.3 IPTables

Netfilter/IPTables 体系结构由两部分组成，一部分是 Netfilter 的钩子函数，另一部分则是指导这些钩子函数如何工作的一系列规则，这些规则存储在被称为表（table）的数据结构之中。钩子函数通过访问表中的规则来判断应该返回什么值给 Netfilter 模块。IPTables 组件包含这些表以及对这些表进行管理的命令 iptables。本章的扩展与提高一节将讨论如何通过 iptables 命令来对这些表进行操作。

1. 内核内建表

内核默认建立 3 个表：filter 表、nat 表和 mangle 表，绝大部分数据报处理功能都可以通过在这些内建的表中填入规则来完成。

（1）filter 表

filter 表用于过滤数据报，根据事先设定好的规则来判断是接受还是拒绝数据报。它在 NF_IP_LOCAL_IN、NF_IP_FORWARD 和 NF_IP_LOCAL_OUT 3 处注册了钩子函数。即 filter 表将数据报进入本地进程之前，数据报在内核通过路由算法即将被转发之前以及本地进程向网络发送数据报时发挥作用。

（2）nat 表

nat 表用于网络地址转换（Network Address Translation，NAT）。nat 表在 NF_IP_PRE_ROUTING 和 NF_IP_POST_ROUTING 两处注册了钩子函数。如果需要，它还可以在 NF_IP_LOCAL_IN 和 NF_IP_LOCAL_OUT 两处注册钩子，提供本地数据报的地址转换功能。

（3）mangle 表

mangle 表用于对指定数据报进行修改，可供修改的数据报内容包括 MARK、TOS、TTL、SECMARK 和 CONNSECMARK 这些字段。mangle 表的钩子函数嵌入在 Netfilter 的 NF_IP_PRE_ROUTING、NF_IP_LOCAL_IN、NF_IP_FORWARD、NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 等 5 处检查点，即 mangle 表可以在所有检查点注册钩子函数。

网卡接收数据报后，数据报在内核中经过的表的路线如图 11-4 所示。

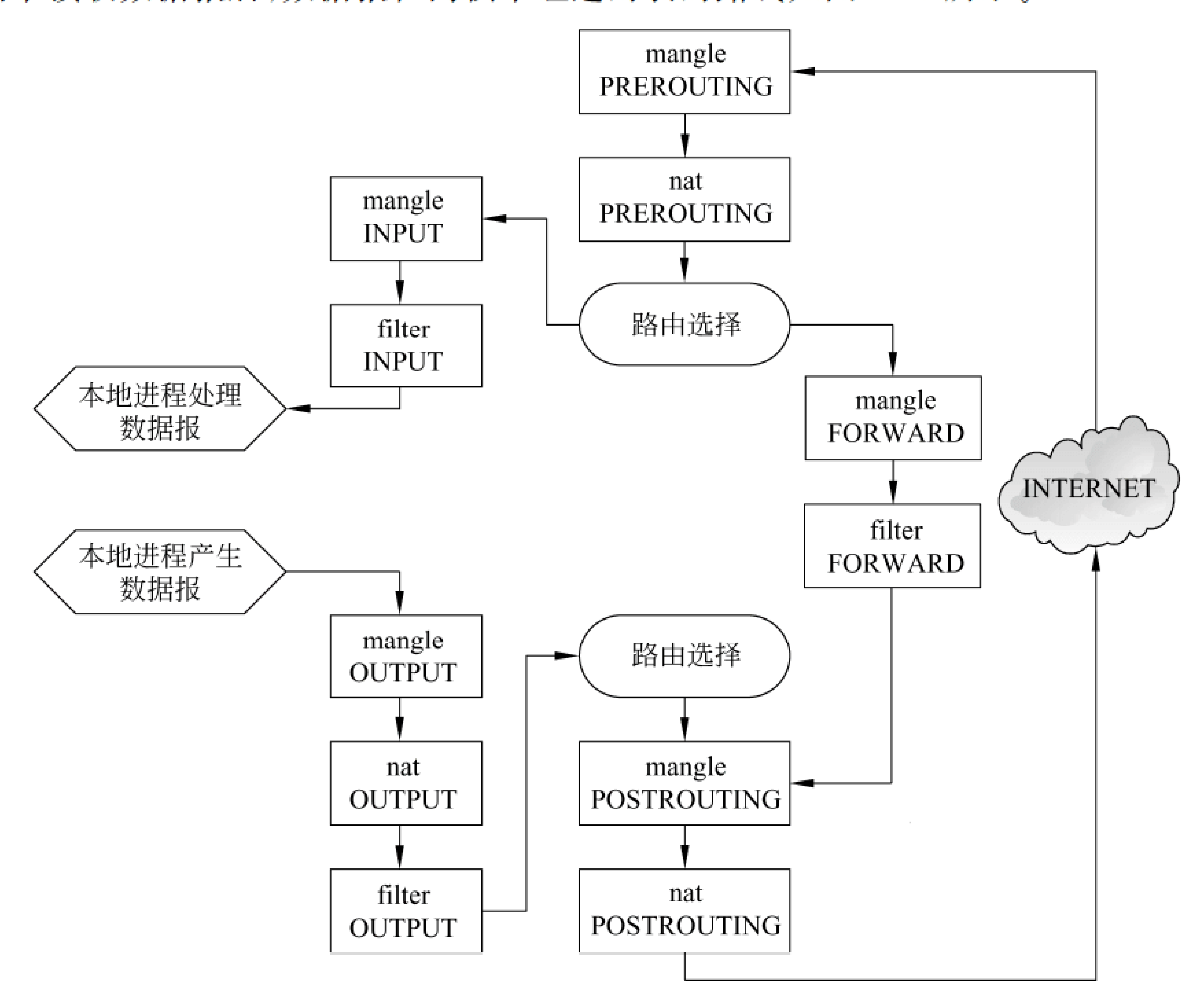


图 11-4　数据报途经的表的路线示意图

内核编程人员还可以通过注入模块的方式，调用 Netfilter 的接口函数创建新的表。Linux 内核中，表的定义如下：

```
struct ipt_table
{
    struct list_head list;                    //表链
    char name[IPT_TABLE_MAXNAMELEN];          //表名，如"filter"、"nat"等
    struct ipt_replace * table;               //表模子，初始为 initial_table.repl
    unsigned int valid_hooks;                 //位向量，标示本表所影响的 hook
    rwlock_t lock;                            //读写锁，初始为打开状态
    struct ipt_table_info * private;          //表的数据区
    struct module * me;                       //是否在模块中定义
}
```

其中关键的数据结构为 ipt_table_info 结构，该结构描述了表的数据结构，包括表的大小，表中规则数等信息。其在内核中的定义如下：

```
struct ipt_table_info
{
    unsigned int size;                                  //表的大小
    unsigned int number;                                //表中的规则数目
    unsigned int initial_entries;                       //初始规则数
    unsigned int hook_entry[NF_IP_NUMHOOKS];
    //记录所影响的 hook 的规则入口相对于后面 entries 这个参数的偏移量
    unsigned int underflow[NF_IP_NUMHOOKS];
    //与 hook_entry 相对应的规则表上限偏移量，当无规则录入时，相应的 hook_entry 和
    //underflow 均为 0
    char entries[0]_cacheline_aligned;                  //规则表的入口
};
```

通过 Netfilter 模块构建防火墙最常见的操作就是在 filter 表中设定一系列的规则（rules），从而实现对数据报过滤的操作。在 IPTables 组件中，链（chains）是规则（rules）的集合。filter 表包含 3 个链，分别为 INPUT，FORWARD 和 OUTPUT。该 3 条规则链将分别在数据报在进入本地进程之前、数据报在内核通过路由算法即将被转发之前和本地进程向网络发送数据报前 3 个关键位置发挥作用。内核内建的 filter 表的初始定义如下：

```
static struct ipt_table packet_filter
={
    {NULL, NULL},                   //链表
    "filter",                       //表名
    &initial_table.repl,            //初始的表模板
    FILTER_VALID_HOOKS,             //位向量
    RW_LOCK_UNLOCKED,               //锁
    NULL,                           //初始的 private 数据区为空
    THIS_MODULE                     //模块标示
};
```

其中 FILTER_VALID_HOOKS 的定义为：((1≪NF_IP6_LOCAL_IN)|(1≪NF_IP6_FORWARD) | (1≪NF_IP6_LOCAL_OUT))，即表明 filter 表只在 INPUT、FORWARD 和 OUTPUT 这 3 个检查点生效。初始的 private 数据区为空，在调用 ipt_register_table(&packet_filter)后，filter 表的 private 数据区会按照设定的模板填充好。

2. 规则

IPTables 中每一个表有 0 到多条链，每一条链是一系列规则的集合。每一条规则由两部分组成，第一部分包含 0 个或多个过滤条件(match)，其作用是检查包是否符合过滤条件(所有的条件都必须成立才生效)，第二部分称为目标(target)，用于决定如何处置符合过滤条件的数据报。对每一条规则，IPTables 维护两个计数器：一个计算符合条件的数据报数(packet counter)；另一个计算该规则所处理的总字节数(byte counter)。每当有数据报符合特定规则的过滤条件时，该规则的 packet counter 便会被累加 1，并将该数据报的大小累加到该规则的 byte counter 中。

规则可以只包含过滤条件和目标的一部分。没有设定过滤条件时，则所有数据报都算符合条件。没有设定目标时，则默认让数据报继续其流程，即将数据报转入 TCP/IP 协议栈进行正常处理，而不会对数据报进行任何其他动作，只是该规则的两个计数器会增加计数而已。在 Linux 内核中，规则用 struct ipt_entry 结构表示，主要字段包括匹配用的 IP 头部、0 个或多个过滤条件以及一个目标。由于过滤条件数目不定，所以一条规则实际占用的空间是可变的。ipt_entry 结构定义如下：

```
struct ipt_entry
{
    struct ipt_ip ip;                    //所要匹配的数据报的 IP头信息
    unsigned int nfcache;                //位向量,表示本规则关心数据报的什么部分
    u_int16_t target_offset;             //target 区的偏移
    u_int16_t next_offset;               //下一条规则相对于本规则的偏移
    unsigned int comefrom;               //位向量,标记调用本规则的 hook 号
    struct ipt_counters counters;        //记录该规则处理过的数据报数和数据报总字节数
    unsigned char elems[0];              //target 或者是 match 的起始位置
}
```

其中 target_offset 为 target 区域的偏移，一般 target 区域位于 match 区域之后，而 match 区域则在 ipt_entry 的末尾。target_offset 初始化为 sizeof(struct ipt_entry)，即初始化时假定是没有 match 的。next_offset 为下一条规则相对于本规则的偏移，也即本规则所用空间的总和，初始化为 sizeof(struct ipt_entry)+sizeof(struct ipt_target)，同样也是假定没有 match 的。

3. 过滤条件

iptables 命令用于设置多种过滤条件，但是某些条件需要内核支持相关功能才行。iptables 命令默认会设置一个一般性的 IP 包过滤条件。因此即使没有载入任何扩充模块，用户也可以用 IP 包头的源 IP 地址、目的 IP 地址和协议类型等字段作为过滤条件。除了 IP

之外的其他协议(如 ICMP、TCP 和 UDP 等)必须载入相关的扩充模块,才可以作为过滤条件。可以通过 iptables 的-m 或--match 选项来载入特定的模块。

4. 目标

目标(target)决定了如何处理符合过滤条件的数据报,当满足过滤条件时对该数据报应采取什么样的动作,如接收、丢弃等。IPTables 组件默认有 4 种目标(如表 11-1 所示)。如果想要加入其他的目标,必须通过扩充模块来加入。表 11-1 列出的只是 IPTables 内建的目标。

表 11-1 IPTables 内建的目标

目　标	说　明
ACCEPT	接收数据报,处理后不再去匹配其他的规则,直接跳往下一个规则链
DROP	丢弃数据报,处理后将不再匹配其他的规则,直接中断过滤程序
QUEUE	将数据报传入用户空间进行处理
RETURN	结束当前规则链中的过滤程序,返回主规则链,继续过滤

规则都挂接在各自表的相应 hook 的入口处,当数据报流经该 hook 时,对各个规则进行匹配,对于与某个规则匹配成功的数据报,调用该规则对应的 target 来处理。

11.2.4 Netfilter 内核模块扩充

1. Linux 内核模块机制

操作系统内核主要分为微内核和单一内核两种体系结构。微内核操作系统的内核很小,只实现最基本的服务,如内存管理、进程管理等。而网络协议、文件系统都是在内核的外层实现的。这样做的优点是内核小,层次结构非常清楚,比较方便扩展。缺点是层与层之间的信息交换使得系统的运行效率较低。在单一内核的操作系统中,内存管理、进程管理、网络协议、文件系统等都是在内核中实现的,运行速度非常快,但是由于其所有的内容都集成在内核中,所以可扩展性和可维护性较差。WindowNT 就是微内核的体系结构,Linux 操作系统属于单一内核的体系结构。为了弥补单一内核体系结构可扩展性差的这一缺陷,Linux 操作系统提供了内核模块机制,内核模块是可以动态加载到内核空间运行的程序,模块机制的出现使得用户无须再重新编译整个内核就能挂载和卸载某些功能,从而实现内核的动态扩展。

内核模块机制是 Linux 内核向外部提供的一个接口,其全称为动态可加载内核模块(Loadable Kernel Module,LKM)。模块是具有独立功能的程序,它可以被单独编译,但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间中运行,这与运行在用户空间中的进程不同。

用户编写内核模块时,在编写的模块中必须提供两个函数:init_module()以及 cleanup_module()函数。这两个函数在加载和删除模块时将会被调用。其函数原型分别如下:

```
int init_module(void);
```

```
void cleanup_module(void);
```

内核模块源程序编译成.ko 文件后(2.6 内核是.ko,2.4 内核以及之前是.o)就可以将其加载到内核中。insmod 命令用于将模块加载到内核中。例如 insmod filter.ko 就是将 filter 这个模块加载到内核中。通过 lsmod 命令可以看到加载的 filter 这个模块的名字。如果要将 filter 这个模块删除,只要使用 rmmod filter.ko 即可。insmod 在加载模块时,会调用模块中的 init_module()函数,如果传回 0 则表示成功,模块会被加载;否则表明加载失败。

2. Netfilter 内核模块

Netfilter 为每种网络协议定义了一套钩子函数(hook),这些钩子函数在数据报流过协议栈的几个检查点时被调用。Linux 内核网络堆栈维护一个全局的二维数组来存储这些钩子函数。该二维数组定义为:struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS],其中第一维用于指定协议族,第二维用于指定 hook 的类型。nf_hooks 数组中的元素定义如下:

```
struct nf_hook_ops
{
    struct list_head list;          //链表
    nf_hookfn * hook;               //钩子函数指针
    int pf;                         //协议号
    int hooknum;                    //hook 号
    int priority;                   //优先级,在 nf_hooks 链表中各处理函数按优先级排序
};
```

nf_hooks 数组本质上就是一个类型为 nf_hookfn 的函数指针数组。对数据报的处理就是通过调用这些函数指针所指向的函数来进行的。注册一个 Netfilter 钩子函数实际上就是在由协议族和 hook 类型确定的钩子函数链表中添加一个新的节点。

(1) 注册钩子函数

在 Netfilter 模块中向内核注册钩子函数时要用到以下两个函数:

① nf_register_hook(struct nf_hook_ops * reg)

注册函数,用来向内核注册自定义的钩子函数,如果注册成功,则返回 0;如果失败,则返回非 0 值。

② nf_unregister_hook(struct nf_hook_ops * reg)

卸载函数,用来卸载已经注册的钩子函数。

其中参数 nf_hook_ops 结构中有一个 nf_hookfn 结构的成员,即钩子函数指针。

(2) 钩子函数的原型

钩子函数的原型如下:

```
typedef unsigned int nf_hookfn(unsigned int hooknum,struct sk_buff * * skb,const struct net_device * in,
const struct net_device * out,int (* okfn)(struct sk_buff * ))
```

① 参数 hooknum 用于指定 hook 类型。

② 参数 skb 是一个指向指针的指针,该指针指向的指针指向一个 sk_buff 数据结构,

Linux 网络协议栈中用 sk_buff 这一数据结构来描述数据报。后面将详细介绍该结构。

③ 参数 in 用于指定数据报到达的网络接口。

④ 参数 out 用于指定数据报出去的网络接口。out 和 in 都是 net_device 结构的指针，net_device 结构的定义在文件 linux/netdevice.h 中。Linux 用该结构来描述网络接口。一般情况下，函数只会提供 out 和 in 这两个参数中的其中之一。例如：参数 in 只用于 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_IN 类型的钩子函数，而参数 out 只用于 NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 类型的钩子函数。

⑤ 函数的最后一个参数是一个名为 okfn 的函数指针，该函数以一个 sk_buff 数据结构作为它唯一的参数，并且返回一个整型值。其作用就是如果没有注册任何钩子函数，Netfilter 将调用该函数对数据报进行后续处理。

(3) sk_buff 结构

Linux 网络协议栈中用 sk_buff 这一数据结构来描述数据报。本章实现的防火墙程序也是根据该结构中的一些字段来设置过滤条件，从而进行数据报过滤的。该数据结构在文件 linux/skbuff.h 中进行了定义(代码如下)。

```
struct sk_buff {
    //These two members must be first.
    struct sk_buff          * next;
    struct sk_buff          * prev;
    struct sock             * sk;
    struct skb_timeval      tstamp;
    struct net_device       * dev;
    struct net_device       * input_dev;
    union {
        struct tcphdr       * th;
        struct udphdr       * uh;
        struct icmphdr      * icmph;
        struct igmphdr      * igmph;
        struct iphdr        * ipiph;
        struct ipv6hdr      * ipv6h;
        unsigned char       * raw;
    } h;
    union {
        struct iphdr        * iph;
        struct ipv6hdr      * ipv6h;
        struct arphdr       * arph;
        unsigned char       * raw;
    } nh;
    union {
        unsigned char       * raw;
    } mac;
    ...
};
```

sk_buff 数据结构中最常用的部分就是 h、nh 与 mac 的联合(union)。这 3 个联合分别用于描述传输层包头(如 UDP、TCP 及 ICMP)、网络层包头(如 IPv4/6、ARP)以及链路层包头。每个联合都包含了几个结构，具体哪个结构起作用依赖于具体数据报中使用的协议。在程序中可以通过 sk_buff 指针，定位到数据报的头部，然后通过分析数据报的头部字段对数据报进行过滤。

11.3　实例编程练习

11.3.1　编程练习要求

上一节介绍了 Netfilter 和内核模块的相关知识，接下来通过实现 3 个示例程序来说明如何对 Netfilter 内核模块进行扩展编程。第 1 个程序实现基于协议过滤数据报的功能，第 2 个程序实现基于源 IP 地址过滤数据报的功能，第 3 个示例实现基于目的端口过滤 TCP 包的功能。这 3 个示例程序的主体架构基本一样，主要的区别在于其使用的钩子函数。程序通过钩子函数对数据报实行不同的操作从而实现不同方式的过滤。

11.3.2　编程训练设计与分析

1. 基于协议过滤的示例程序

该程序要实现的功能是根据传输层协议来进行数据报的过滤。即拒绝 ICMP 包，只允许 TCP 包和 UDP 包通过。

(1) 程序中的钩子函数

该程序中的钩子函数定义如下：

```
unsigned int hook_func(unsigned int hooknum, struct sk_buff * * skb, const struct net_device * in,const
struct net_device * out,int (* okfn)(struct sk_buff * ))
    {
        struct sk_buff * pskb= * skb;
        switch(pskb->nh.iph->protocol)
        {
    case IPPROTO_ICMP:
        {
            printk("ICMP Packet:DROP\n");
            return NF_DROP;
        }
    case IPPROTO_TCP:
        {
            printk("TCP Packet:ACCEPT\n");
            return NF_ACCEPT;
        }
    case IPPROTO_UDP:
        {
```

```
                printk("UDP Packet:ACCEPT\n");
                return NF_ACCEPT;
            }
        default:
            {
                printk("Unknown Packet:DROP\n");
                return NF_DROP;
            }
            }
        }
```

根据前面小节介绍的关于 sk_buff 结构的知识,程序可以通过一个 sk_buff 指针来定位数据报的 IP 头部,然后根据该头部的协议字段进行数据报的过滤。

在内核编程中,不能使用用户态 C 语言库函数中的 printf()函数来输出信息,而只能使用 printk()函数。但是尽管使用 printk()函数来输出信息,在控制台也不能看到输出的信息。这是因为内核中 printk()函数的设计目的并不是为了和用户交流,它实际上是内核的一种日志机制,是用来记录日志信息或者给出警告提示的。每个 printk 都会有个优先级,内核一共有 8 个优先级,它们都有对应的宏定义。如果未指定优先级,内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。如果 printk 的优先级比当前终端的优先等级高,消息就会打印到控制台上。因此如果想要在控制台看到 printk()函数的输出信息,可以设置一下 printk()函数的优先级,使其比当前终端的优先等级高,就可以向终端上输出信息了。如果 syslogd 和 klogd 守护进程在运行,则不管是否向控制台输出,消息都会被追加进/var/log/messages 文件中。所以可以打开/var/log/messages 这个文件来查看输出的信息,或者可以直接通过 dmesg 命令来查看输出的信息。第一个程序没有对 printk()函数的优先级进行设置,因此在控制台看不到输出的信息。后面的两个示例程序对 printk()函数的优先级进行了设置,使其可以向控制台输出信息。

(2) 程序中的 init_module()函数

```
int init_module()
{
    nfho.hook       =hook_func;             //hook function
    nfho.hooknum    =NF_IP_PRE_ROUTING;     //the hook point
    nfho.pf         =PF_INET;
    nfho.priority   =NF_IP_PRI_FIRST;       //priority
    nf_register_hook(&nfho);                //register the hook
    return 0;
}
```

nfho 是一个 nf_hook_ops 结构的变量,该函数首先对 nfho 的一些成员赋值,其中 hook_func 就是程序中定义的钩子函数,NF_IP_PRE_ROUTING 指定了该钩子函数所在的检查点,PF_INET 指定协议族,NF_IP_PRI_FIRST 指定了该钩子函数的优先级。最后调用 nf_register_hook 函数向内核注册钩子函数。

(3) 程序中的 cleanup_module()函数(代码如下)

```
void cleanup_module()
{
    nf_unregister_hook(&nfho);
}
```

该函数调用 nf_unregister_hook 来卸载已经注册的钩子函数。

（4）Makefile 文件（代码如下）

```
obj-m:=filter_prot.o
KDIR:=/lib/modules/$(shell uname-r)/build
PWD:=$(shell pwd)
all:
$(MAKE)-C $(KDIR) SUBDIRS=$(PWD) modules
clean:
$(MAKE)-C $(KDIR) SUBDIRS=$(PWD) clean
```

代码经过编译后会生成一个 filter_prot.ko 文件，然后就可以通过 insmod filter_prot.ko 来加载该模块，通过 rmmod filter_prot.ko 来卸载该模块。加载该模块之前，其他主机可以 ping 通本机，但是加载该模块之后其他主机将 ping 不通本机。因为所有的 ping 本机的 ICMP 包都被直接丢弃了。测试如下。用某台主机一直 ping 本机，然后通过 dmesg 命令查看日志消息，可以看到内核日志消息中有如下内容（注意，所有的 ICMP 包均被丢弃了）：

```
[270614.435983] ICMP Packet:DROP
[270619.923230] ICMP Packet:DROP
[270627.964276] TCP Packet:ACCEPT
[270627.989874] TCP Packet:ACCEPT
[270632.174701] UDP Packet:ACCEPT
```

2. 基于源 IP 地址过滤的示例程序

该程序要实现的功能就是对源 IP 地址为“192.168.1.27”的数据报全部丢弃。程序的钩子函数定义如下：

```
unsigned int hook_func(unsigned int hooknum, struct sk_buff * * skb,const struct net_device * in,const
struct net_device * out,int(* okfn)(struct sk_buff * ))
{
    struct sk_buff * pskb= * skb;
    if((pskb->nh.iph->saddr)==in_aton("192.168.1.27"))
    {
        printk("<0>""A Packet from 192.168.1.1:DROP\n");
        return NF_DROP;
    }
    else
    {
        return NF_ACCEPT;
    }
```

```
}
```

和上一个程序的主要区别在于在该钩子函数中，程序首先根据 sk_buff 指针定位数据报的 IP 头，然后根据 IP 头的源 IP 地址过滤数据报。注意到程序中用 in_aton()这个函数将一个字符串形式的 IP 转换为一个长整型的值。该函数是 inet_addr()函数在内核中的替代函数。因为是内核编程，所以程序不能直接使用 C 语言库函数中的 inet_addr()函数。

Makefile 文件和前一个程序类似，编译并加载到内核后进行如下测试。尝试从 IP 地址为“192.168.1.27”的主机 ping 本机，telnet 本机或者通过 ssh 登陆本机。可以看到无论通过什么方式都连不上本机，因为所有来自 192.168.1.27 的数据报都直接被丢弃了。通过 dmesg 命令查看日志可以看到日志中有如下内容：

```
[270805.756903] A Packet from 192.168.1.27:DROP
[270814.551829] A Packet from 192.168.1.27:DROP
[270819.723446] A Packet from 192.168.1.27:DROP
[270825.205202] A Packet from 192.168.1.27:DROP
[270838.713382] A Packet from 192.168.1.27:DROP
```

3. 基于 TCP 通信目的端口过滤的示例程序

该程序要实现的功能就是对于目的端口地址为“23”的 TCP 包全部丢弃。程序的钩子函数定义如下：

```
unsigned int hook_func(unsigned int hooknum, struct sk_buff * * skb, const struct net_device * in, const
struct net_device * out, int (* okfn)(struct sk_buff * ))
{
    struct sk_buff * pskb= * skb;
    struct tcphdr * thdr= (struct tcphdr * )(pskb->data+ (pskb->nh.iph->ihl * 4));
    if((pskb->nh.iph->protocol)!= IPPROTO_TCP)
    {
        printk("<0>""Not A TCP Packet:ACCEPT\n");
        return NF_ACCEPT;
    }
    else
    {
        if(thdr->dest= = in_pton("23"))
        {
            printk("<0>""A TCP Packet PORT 23:DROP\n");
            return NF_DROP;
        }
        else
            return NF_ACCEPT;
    }
}
```

在该函数中，程序首先根据协议类型判断是否是 TCP 包，然后通过 sk_buff 的 IP 头计算出 TCP 头部位置并定义一个 TCP 头部指针指向该位置。最后通过该 TCP 头部指针取

得该 TCP 包的目的端口进行过滤。

因为 TCP 包头部存的端口是网络序，所以在进行判断时，需要将“23”这个数值也转换成网络序。转换函数如下：

```
unsigned short in_pton(const char * port_str)
{
    unsigned short p,i;
    for(p=0,i=0;port_str[i]<='9'&&port_str[i]>='0';i++ )
    {
        p=p*10+ (port_str[i]-'0');
    }
    i= (p>>8)& 0x0000ff;
    i |= (p<< 8)& 0x00ff00;
    return(i);
}
```

编译并加载到内核后进行如下测试。尝试从其他主机 telnet 本机，查看日志可以看到通过 23 端口进来的数据报都被直接丢弃掉了。

```
[270977.326040]A TCP Packet PORT 23:DROP
[270980.327817]A TCP Packet PORT 23:DROP
[270986.223673]A TCP Packet PORT 23:DROP
```

11.4　扩展与提高

11.4.1　iptables 命令

IPTables 组件可以通过 iptables 命令在用户空间对 netfilter 内核模块中表的规则进行插入、删除和修改。发行较早的 Linux 版本可能没有将该组件包含在内，需要从 netfilter.org 官方网站上下载该工具另行安装。

通过 iptables 命令，读者可以很方便地设定规则从而快速地定制自己的防火墙，这些规则存储在内核空间的表中。每条规则都有自己的目标，它们告诉内核如何对数据报进行处理。如果某个数据报与规则匹配，则可以使用目标 ACCEPT 允许该数据报通过，或者使用目标 REJECT 或 DROP 来阻塞或者丢弃数据报。

11.4.2　iptables 命令参数详解

语法：

```
iptables[-t table]command[match][-j target/jump]options
```

table 参数用来指定规则表，内核默认内建 3 个 table：filter、nat 和 mangle，当参数没有指定某个规则表时，默认是对 filter 表进行操作。其中各个表的作用可以参考前面小节的介绍。

(1) command 指定操作的命令,如插入,删除规则等。
(2) match 设定匹配参数。
(3) target 指定对数据报的处理动作(例如:DROP、LOG、ACCEPT 或 REJECT)。

1. 常用命令

(1) 命令-A,--append
示例:

```
iptables -A INPUT
```

说明:新增规则到某个规则链中,该规则将会成为规则链中的最后一条规则。当源地址或目的地址以名字而不是 IP 地址的形式出现时,若这些名字可以被解析为多个地址,则这条规则会和所有可用的地址结合。
(2) 命令-D,--delete
示例:

```
iptables -D INPUT --dport 80 -j DROP
```

或者

```
iptables -D INPUT 1
```

说明:从指定链中删除规则。有两种方法指定要删除的规则:一是输入完整的规则,另一个是指定规则在所选链中的序号(每条链的规则都各自从 1 被编号)。
(3) 命令-R, --replace
示例:

```
iptables -R INPUT 1 -s 192.168.1.1 -j DROP
```

说明:在指定链中指定的行上(每条链的规则都各自从 1 被编号)替换规则,规则被取代后并不会改变顺序。它主要的用处是试验不同的规则。当源地址或目的地址以名字而不是 IP 地址的形式出现时,若这些名字可以被解析为多个地址,则这条 command 会失败。
(4) 命令-I,--insert
示例:

```
iptables -I INPUT 1 --dport 80 -j ACCEPT
```

说明:插入一条规则,原本该位置上的规则将会往后移动一个顺位。如果序号为 1,则规则会被插入链的头部。
(5) 命令-L, --list
示例:

```
iptables -L INPUT
```

说明:列出某规则链中的所有规则。如果没有指定链,则显示指定表中的所有链。如果什么都没有指定,就显示默认表中的所有的链。精确输出会受其他参数影响,如-n 和-v 等参数,后面会介绍这些参数。

（6）命令-F，--flush

示例：

```
iptables -F INPUT
```

说明：清空指定的链。如果没有指定链，则清空指定表中的所有链。如果什么都没有指定，就清空默认表中所有的链。

（7）命令-Z，--zero

示例：

```
iptables -Z INPUT
```

说明：将指定链的计数器归零。数据报计数器用来统计同一数据报出现的次数，是过滤阻断式攻击非常重要的工具。

（8）命令-N，--new-chain

示例：

```
iptables -N allowed
```

说明：根据用户指定的名字建立新的链。注意，新建链的名字不能和已有的链、target 同名。

（9）命令-X，--delete-chain

示例：

```
iptables -X allowed
```

说明：删除指定的用户自定义链。这个链必须没有被引用，如果被引用，在删除之前必须删除或者替换与之有关的规则。如果没有给出参数，这条命令将会删除默认表中所有非内建的链。

（10）命令-P，--policy

示例：

```
iptables -P INPUT DROP
```

说明：对指定链设置默认策略。所有不符合规则的包都会被强制使用这个策略。注意只有内建的链才可以使用规则。

（11）命令-E，--rename-chain

示例：

```
iptables -E allowed disallowed
```

说明：对自定义的链进行重命名，原来的名字在前，新名字在后。注意：这仅仅只是改变链的名字，而对整个表的结构、功能没有任何影响。

2. 常用数据报匹配参数

（1）参数-p，--protocol

示例：

```
iptables -A INPUT -p tcp
```

说明：匹配通信协议类型，可以使用!运算符进行反向匹配，例如：-p!tcp ，意思是匹配除 tcp 以外的其他协议类型，例如 udp、icmp 等。如果要匹配所有类型，则可以使用 all 关键字来进行匹配，例如：iptables -A INPUT -p all。

(2) 参数-s，--src，--source

示例：

```
iptables -A INPUT -s 192.168.1.1
```

说明：匹配数据报的源 IP 地址，可以匹配单个 IP 地址或某网段的 IP 地址，匹配网段 IP 地址时用数字来表示屏蔽，例如：-s 192.168.0.0/24，匹配 IP 时可以使用!运算符进行反向匹配，例如：-s!192.168.0.0/24。

(3) 参数-d，--dst，--destination

示例：

```
iptables -A INPUT -d 192.168.1.1
```

说明：匹配数据报的目的 IP 地址，具体使用方式与-s 参数的使用方式相同。

(4) 参数-i，--in-interface

示例：

```
iptables -A INPUT -i eth0
```

说明：匹配数据报进入的网卡，可以使用通配字符＋实现多个网卡的匹配，例如：-i eth＋表示所有的 ethernet 网卡，也以使用!运算符进行反向匹配，例如：-i!eth0。

(5) 参数-o，--out-interface

示例：

```
iptables -A FORWARD -o eth0
```

说明：匹配数据报出去的网卡，具体使用方式和-i 参数的使用方式相同。

(6) 参数--sport，--source-port

示例：

```
iptables -A INPUT -p tcp --sport 22
```

说明：匹配数据报的源端口号，可以匹配单一端口号或者指定的某个范围内所有的端口号，例如：--sport 22:80，表示从 22 到 80 之间的端口号都匹配，如果要匹配不连续的多个端口号，则必须使用--multiport 参数，后面将会介绍。匹配端口号时，同样可以使用!运算子进行反向匹配。

(7) 参数--dport，--destination-port

示例：

```
iptables -A INPUT -p tcp --dport 22
```

说明：匹配数据报的目的端口号，具体使用方式与--sport 参数的使用方式相同。

(8) 参数--tcp -flags

示例：

```
iptables -p tcp --tcp-flags SYN,FIN,ACK SYN
```

说明：匹配 TCP 数据报的标志位，参数分为两部分，第一部分列出想匹配的标志位，第二部分则列出需要设置的标志位。TCP 包标志位包括：SYN(同步)、ACK(应答)、FIN(结束)、RST(重设)和 URG(紧急)等。除此之外还可以使用关键词 ALL 和 NONE 进行匹配。同样也可以使用!运算符进行反向匹配。

(9) 参数-syn

示例：

```
iptables -p tcp -syn
```

说明：匹配用于连接的 TCP 数据报，其作用相当于 iptables-p tcp --tcp-flags SYN，FIN，ACK SYN。

(10) 参数-m multiport --source -port

示例：

```
iptables -A INPUT -p tcp -m multiport --source-port 22,53,80,110
```

说明：匹配不连续的多个源端口号，一次最多可以列 15 个端口号，可以使用!运算符进行反向匹配。

(11) 参数-m multiport --destination-port

示例：

```
iptables -A INPUT -p tcp -m multiport --destination-port 22,53,80,110
```

说明：匹配不连续的多个目的端口号，具体使用方式和-m multiport --source-port 参数的使用方式相同。

(12) 参数-m multiport -port

示例：

```
iptables -A INPUT -p tcp -m multiport --port 22,53,80,110
```

说明：匹配源端口和目的端口号都相同的数据报，设定方式同上，具体使用方式和-m multiport --source-port 参数的使用方式相同。值得注意的是源端口和目的端口号都相同的数据报才符合匹配条件。例如源端口号为 80，目的端口号为 110，这种数据报就不符合条件。

(13) 参数--icmp-type

示例：

```
iptables -A INPUT -p icmp --icmp-type 8
```

说明：匹配 ICMP 的类型编号，可以使用代码或数字编号进行匹配。可以输入 iptables -p icmp --help 查看有哪些代码可用。

(14) 参数-m limit -limit

示例：

```
iptables -A INPUT -m limit --limit 3/second
```

说明：匹配某段时间内数据报的平均流量，上面的示例是用来匹配：每秒平均流量是

否超过 3 个封包。此外还可以按每分钟、每小时或每天平均一次，默认值为每小时平均一次，所对应的参数分别是：/minute、/hour、/day。除了进行数据报数量的匹配外，设定这个参数也会在条件达成时，暂停对数据报的匹配，从而避免恶意攻击者使用洪水攻击，导致服务被阻断。

(15) 参数--limit-burst

示例：

```
iptables -A INPUT -m limit --limit-burst 5
```

说明：匹配瞬间大量数据报的数量，上面示例是用来匹配一次同时涌入的数据报是否超过 5 个，超过此上限的数据报将被直接丢弃。

(16) 参数-m mac --mac-source

示例：

```
iptables -A INPUT -m mac --mac-source a3:3d:ee:6a:4c:01
```

说明：匹配数据报的源 MAC 地址。

(17) 参数-m owner --uid-owner

示例：

```
iptables -A OUTPUT -m owner --uid-owner 500
```

说明：匹配来自本机的数据报是否为某特定使用者产生的，这样可以避免服务器使用 root 或其他身份将敏感数据传出。

(18) 参数-m owner --gid-owner

示例：

```
iptables -A OUTPUT -m owner --gid-owner 0
```

说明：匹配来自本机的数据报是否为某特定用户组产生的，使用方式和-m owner --uid-owner 参数类似。

(19) 参数-m owner --pid-owner

示例：

```
iptables -A OUTPUT -m owner --pid-owner 78
```

说明：匹配来自本机的数据报是否为某特定行程产生的，使用方式和-m owner--uid-owner 参数类似。

(20) 参数-m owner --sid-owner

示例：

```
iptables -A OUTPUT -m owner --sid-owner 100
```

说明：匹配来自本机的数据报是否为某特定连接(Session ID)的响应数据报，使用方式和-m owner --uid-owner 参数类似。

(21) 参数-m state-state

示例：

```
iptables -A INPUT -m state --state RELATED,ESTABLISHED
```

说明：匹配会话状态，会话状态共有 4 种：INVALID、ESTABLISHED、NEW 和 RELATED。INVALID 表示该数据报的会话 ID（Session ID）无法辨识或不正确。ESTABLISHED 表示该数据报属于某个已经建立的连接。NEW 表示该数据报想要发起一个连接。RELATED 表示该数据报是属于某个已经建立的连接的。

3. 常用的处理动作

参数-j 用来指定要进行的处理动作，常用的处理动作包括：ACCEPT、REJECT、DROP、REDIRECT、MASQUERADE、LOG、DNAT、SNAT、MIRROR、QUEUE、RETURN 及 MARK 等。常用的处理动作如下：

（1）目标 ACCEPT：接收数据报，进行完该处理后，将不再去匹配其他的规则，而是直接跳往下一条规则链。

（2）目标 REJECT：拦截该数据报，并回送数据报通知对方，可以选择回送 ICMP port-unreachable、ICMP echo-reply 或是 tcp-reset 这几种类型的数据报来通知对方，进行完该处理后，将不再匹配其他的则，而直接中断过滤程序。

（3）目标 DROP：直接丢弃数据报，进行完该处理动作后，将不再匹配其他的规则，而直接中断过滤程序。

（4）目标 REDIRECT：将数据报重定向到另一个端口，进行完该处理动作后，还会继续匹配其他的规则。

（5）目标 MASQUERADE：改写数据报的源 IP 地址为防火墙网卡的 IP 地址，可以指定 port 对应的范围，进行完该处理动作后，直接跳往下一条规则链。该功能与 SNAT 略有不同，当进行 IP 伪装时，不需要指定要伪装成哪个 IP，系统会自动获取网卡的 IP 地址作为伪装 IP。

（6）目标 LOG：将数据报的相关讯息记录在文件/var/log 中。进行完该处理动作后，将会继续匹配其他的规则。

（7）目标 SNAT：改写数据报的源 IP 地址为某特定 IP 或 IP 范围，可以指定 port 对应的范围，进行完该处理动作后，将直接跳往下一条规则。

（8）目标 DNAT：改写数据报的目的 IP 地址为某特定 IP 或 IP 范围，可以指定 port 对应的范围，进行完此处理动作后，将会直接跳往下一条规则链。

（9）目标 MIRROR：对调源 IP 地址与目的 IP 地址，然后将该数据报回送，进行完该处理动作后，将会中断过滤程序。

（10）目标 QUEUE：中断过滤程序，将数据报放入队列，交由其他程序处理。

（11）目标 RETURN：结束当前规则链中的过滤程序，返回主规则链继续过滤。

（12）目标 MARK：数据报标标记，以便给后续过滤的条件提供判断的依据，进行完该处理动作后，将会继续匹配其他的规则。

4. 常用选项

（1）选项-v，--verbose（详细的）

可用此选项的命令是：

```
--list,--append,--insert,--delete,--replace
```

说明：该选项使输出详细化，常与--list 命令一起使用。与--list 连用时，输出中包括网络接口的地址、规则的选项、TOS 掩码、字节和包计数器。如果-v 和--append、--insert、--delete 或--replace 连用，iptables 会输出规则插入或者删除等过程的详细信息。

(2) 选项-x，--exact(精确的)

可用此选项的命令是：

```
--list
```

说明：使--list 输出的计数器显示准确的数值，而不用 K、M、G 等估值。该选项只能和--list 连用。

(3) 选项-n，--numeric(数值)

可用此选项的命令是：

```
--list
```

说明：使输出的 IP 地址和端口以数值的形式显示，而不是默认的名字，比如主机名、网络名等。该选项也只能和--list 连用。

(4) 选项--line-numbers

可用此选项的命令是：

```
--list
```

说明：显示出每条规则在相应链中的序号。该选项也只能和--list 连用。

11.4.3 设计防火墙

根据 11.4.2 节讨论的关于 Netfilter 和 IPTables 的知识，利用 iptables 命令对 Netfilter 进行设置，从而实现一个简单的防火墙。

本节要求实现的防火墙功能包括：

(1) 所有来自 192.168.1.0～192.168.1.254 这个 IP 网段的数据报都设置为接受。

(2) 对于 202.113.25.0～202.113.25.254 这个网段的 IP 数据报，只允许来自 202.113.25.174 这个 IP 地址的数据报通过，其余的都丢弃。

(3) 对 202.113.16.000～202.113.16.254 网段的 IP 的数据报都设定为接受。

(4) 拒绝其他主机通过 ssh 和 telnet 连接本机，即将通过 22 和 23 端口连接本机的数据报全部丢弃。

(5) 允许通过 FTP 连接本机，即将通过 20 和 21 端口连接本机的数据报设置为通过。

1. 清除所有规则

首先清除 filter 表中的规则(代码如下)。

```
iptables -t filter -F          //清空 filter表中的所有链
iptables -t filter -X          //清空 filter表中所有用户的自定义链
iptables -t filter -Z          //把所有链的所有计数器归零
```

也可以不指定 filter 表，当没有指定对某个表进行操作时，默认对 filter 表进行操作。

2. 定义默认的策略

清除 filter 表中所有规则之后，接下来就是对 filter 表中的链设置默认策略。当某个包与所有设定的规则都不匹配时将会被强制使用这个策略。注意只有内建的链才可以设置默认策略，用户自定义的链不能设置默认策略。这里将默认的策略都设置为接受(代码如下)。

```
iptables -t filter -P INPUT ACCEPT
iptables -t filter -P OUTPUT ACCEPT
iptables -t filter -P FORWARD ACCEPT
```

3. 添加规则

按照本节要求实现的防火墙功能来添加自定义的规则。

(1) 所有来自 192.168.1.0～192.168.1.254 这个 IP 网段的数据报都设置为接受(代码如下)。

```
iptables -A INPUT -i eth0 -s 192.168.1.0/24 -j ACCEPT
```

(2) 来自 202.113.25.174 这个 IP 地址的数据报设置为接受(代码如下)。

```
iptables -A INPUT -i eth0 -s 202.113.25.174 -j ACCEPT
```

(3) 对于 202.113.25.0～202.113.25.254 这个网段的 IP 数据报设置为拒绝(代码如下)。

```
iptables -A INPUT -i eth0 -s 202.113.25.0/24 -j DROP
```

第 2 条规则必须在第 3 规则之前，否则第 2 条规则将不会起任何作用。因为当一个来自 202.113.25.174 的数据报进来时，如果第 3 条规则在第 2 条规则之前，按顺序匹配原则，则先匹配第 3 条规则，将该数据报 DROP 掉了，而 Netfilter 在进行完 DROP 这个动作之后将会直接中断过滤程序，不会再继续匹配其他的规则。

(4) 对 202.113.16.000～202.113.16.254 网段的 IP 的数据报都设定为通过(代码如下)。

```
iptables -A INPUT -i eth0 -s 202.113.16.0/24 -j ACCEPT
```

(5) 对于通过 22 和 23 端口连接本机的数据报设置为丢弃(代码如下)。

```
iptables -A INPUT -i eth0 -p TCP --dport 22 -j DROP
iptables -A INPUT -i eth0 -p TCP --dport 23 -j DROP
```

可以将以上两条命令合并成一条(代码如下)：

```
iptables -A INPUT -i eth0 -p TCP --dport 22:23 -j DROP
```

或者写成(代码如下)：

```
iptables -A INPUT -i eth0 -p TCP -m multiport --dport 22,23 -j DROP
```

(6) 对于通过 20 和 21 端口连接本机的数据报设置为通过(代码如下)。

```
iptables -A INPUT -i eth0 -p TCP --dport 20 -j ACCEPT
iptables -A INPUT -i eth0 -p TCP --dport 21 -j ACCEPT
```

同样也可以将以上两条命令合并成一条命令(代码如下):

```
iptables-A INPUT-i eth0-p TCP--dport 20:21-j ACCEPT
```

或者写成(代码如下):

```
iptables-A INPUT-i eth0-p TCP-m multiport--dport 20,21-j ACCEPT
```

至此本节要求实现的防火墙的规则都已经全部设置完成。可以通过命令 iptables-L-n 来查看各个链上设置的规则。以上这些命令的执行都需要 root 权限。该防火墙功能比较简单,只是作为示例来演示如何使用 iptables 命令,读者完全可以根据自己的需要来定制更为实用和功能更强大的防火墙。

第12章

Linux内核网络协议栈加固

12.1 编程训练目的与要求

Linux 是一种开发源代码的操作系统，程序开发人员能够通过修改或升级其源代码对系统进行加固。本章通过加固 Linux 网络协议栈程序，改变 Linux 内核对孤立 TCP SYN 数据包的处理方式，提升系统对 TCP SYN 拒绝服务攻击的防御能力。

本章训练的主要目的是：

(1) 理解 TCP 连接的建立过程以及拒绝服务式攻击的基本原理与方法。

(2) 结合 Linux 内核源代码的代码分析，理解 Linux 网络协议栈的实现原理。

(3) 掌握对 TCP SYN Flood 的防御手段以及对 Linux 内核进行扩展开发的方法。

(4) 了解 Linux TCP cookie 防火墙的工作原理。

本章的训练要求如下：

(1) 扩展 Linux 原有内核功能，使系统能够在遭受 TCP SYN 拒绝服务式攻击后，丢弃 TCP SYN 数据包，从而降低拒绝服务式攻击造成的危害。

(2) 在丢弃新的 TCP SYN 数据包的同时，不能影响系统已经建立的 TCP 连接。

(3) 基于 Linux 2.6 及其以上版本内核代码进行开发。

12.2 相关背景知识

12.2.1 拒绝服务式攻击

拒绝服务式攻击 DoS 是一种简单有效的进攻方式。DoS 耗费攻击目标的系统资源，使攻击目标无法正常提供服务，从而达到攻击的目的。一般攻击者可以利用协议漏洞，模拟众多的服务请求等手段，使目的主机忙于处理各种虚假服务请求，而无法提供正常服务，其中 TCP SYN 洪水攻击就是一种利用 TCP 协议漏洞进行拒绝服务攻击的典型方法。

1. TCP SYN 攻击原理

TCP 建立连接需要通过一个三次握手过程建立起来，所谓三次握手，是指 TCP 协议在发送数据前，TCP 的客户端和服务端通过 3 个数据包，确认双方节点的状态，进而建立连接的过程，具体步骤如下。

(1) 在建立连接前，服务端打开特定端口，进行监听，客服端主动发送 TCP 数据包到服务端主机的该端口上，并将该数据包的“SYN”标志位置位，并初始化发送序列号为 N；

(2) 服务端收到该数据包后,发送回复数据包,该包的接收序列号为 $N+1$,并同时置位“SYN”和“ACK”,初始化该包的发送序列号为 M;

(3) 客户端收到该数据包后,再次发送该数据包的回复数据包,并置位“ACK”,此外,其接受序列号为 $M+1$,至此,一个 TCP 连接建立完成。

三次握手的过程如图 12-1 所示。

如果 Server 端在发出 SYN+ACK 数据包后未能收到 ACK 数据包,就会认为该 SYN+ACK 数据包丢失,进而重新发送该包,直到重复若干次后才确定终止尝试(如图 12-2 所示)。

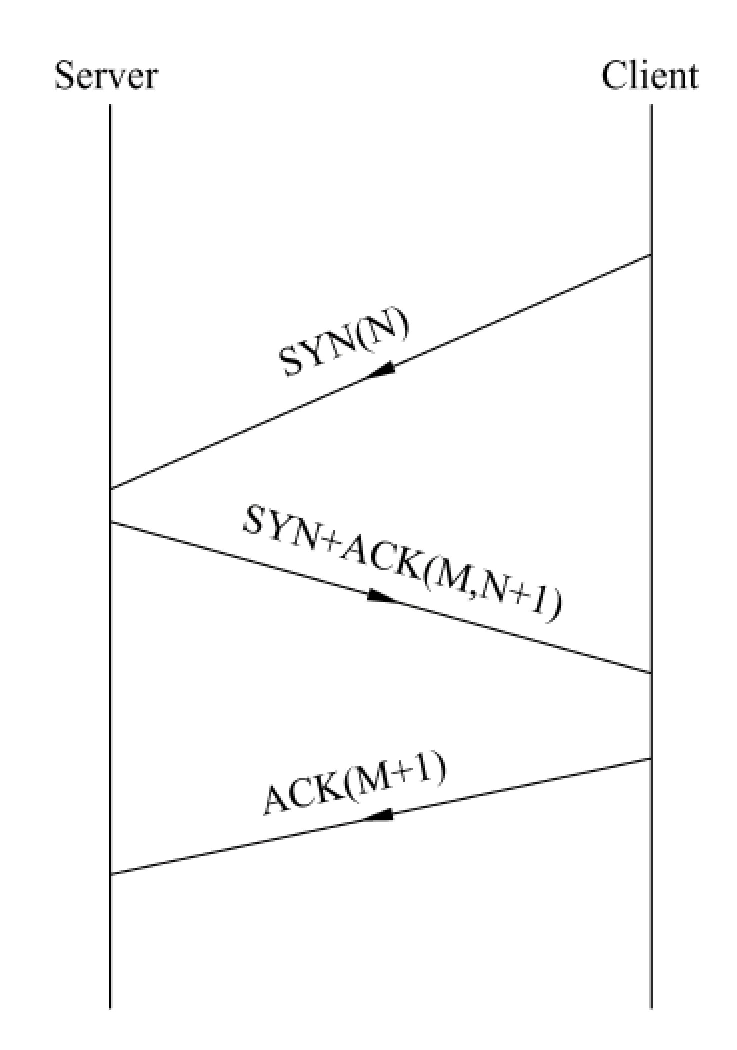


图 12-1　三次握手的过程示意图

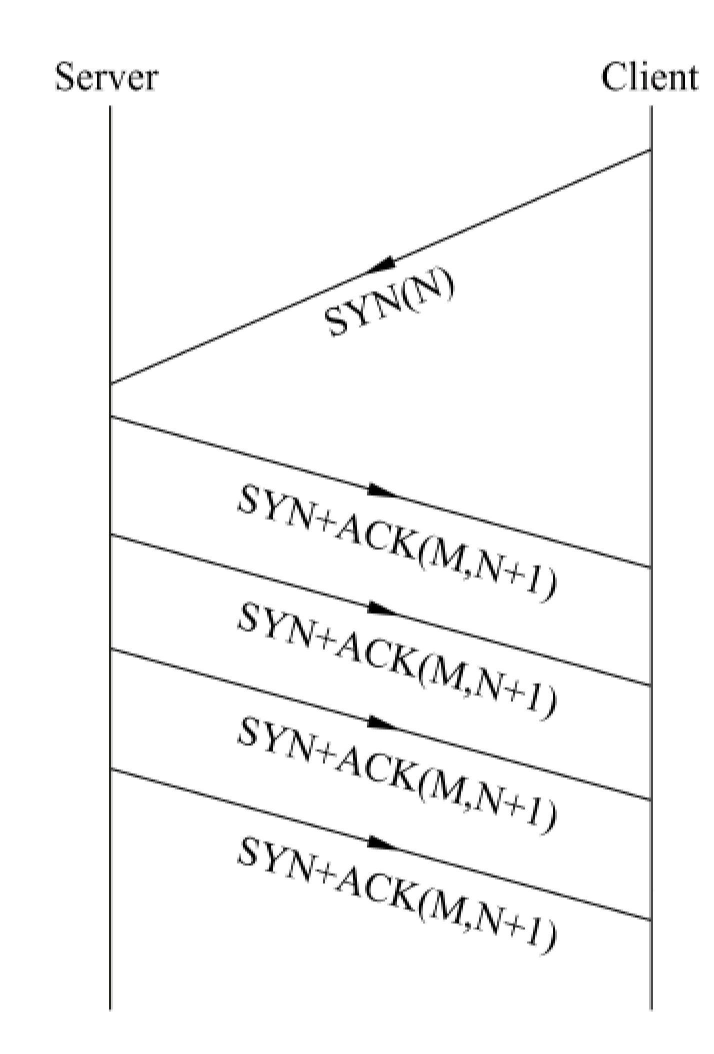


图 12-2　ACK 包丢弃情况示意图

2. TCP SYN 攻击试验

本试验共需要 3 台主机,假设其分别为主机 A、主机 B 和主机 C,3 台主机处于同一局域网内。在本实验中,主机 A 的 IP 地址为 172.20.22.27;主机 B 的 IP 地址为 172.20.22.37,操作系统不限;主机 C 的 IP 地址不限,其上运行 Sniffer 程序,在本书的试验中使用 Iris。为简单起见,主机 B 和主机 C 的角色可由同一台主机扮演。

试验步骤如下:

(1) 在主机 C 安装 Sniffer 程序,并进行监听。为避免干扰,可设置其过滤器使其只监控主机 A、B 之间的 TCP 数据包。

(2) 在主机 A 开放端口 445,并进行监听。

(3) 在主机 B 上使用 Telnet 协议尝试连接该端口,并在主机 C 上使用 Sniffer 软件观测网络数据包的收发情况。

图 12-3 记录了一次完整的 TCP 连接的数据包通信过程。

MAC source addr	MAC dest. addr	Frame	Protocol	Addr. IP src	Addr. IP dest	Port src	Port dest	SEQ	ACK
00:18:8B:00:...	00:1D:09:33...	IP	TCP->MICROSOFT-DS (....S.)	172.20.22.37	172.20.22.27	1254	445	1949739665	0
00:1D:09:33:...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1254	3834677579	1949739666
00:18:8B:00:...	00:1D:09:33...	IP	TCP->MICROSOFT-DS (.A....)	172.20.22.37	172.20.22.27	1254	445	1949739666	3834677580

图 12-3　一次完整的 TCP 连接过程示意图

(4) 在主机 A 发送一个孤立的 TCP SYN 数据包，观察主机 B 的反应，结果如图 12-4 所示。可见，主机 B 在发送 SYN＋ACK 回应数据包后，如果在一定时间内未能收到 ACK 数据包，则会继续重新发送 SYN＋ACK 回应包若干次，直到收到 ACK 数据包或者超时为止。超时时间不同，操作系统的设置也不同。

Time (h:m:s:ms)	MAC source ...	MAC dest. addr	Frame	Protocol	Addr. IP src	Addr. IP dest	Port src	Port dest	SEQ	ACK	Size
13:38:02:984	00:18:8B:00...	00:1D:09:33...	IP	TCP->MICROSOFT-DS (...S.)	172.20.22.37	172.20.22.27	1273	445	286...	0	62
13:38:[illegible]2:984	00:1D:09:33...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1273	126...	286...	62
13:38:[illegible]6:000	00:1D:09:33...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1273	126...	286...	62
13:38:[illegible]2:031	00:1D:09:33...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1273	126...	286...	62
13:38:[illegible]6:359	00:1D:09:33...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1273	126...	286...	62
13:38:[illegible]9:343	00:1D:09:33...	00:18:8B:00...	IP	TCP->MICROSOFT-DS (.A..S.)	172.20.22.27	172.20.22.37	445	1273	126...	286...	62

图 12-4　主机 B 响应数据包示意图

如果主机 A 发送一个“孤立的”SYN 数据包，而主机 B 需要发送若干个 SYN＋ACK 数据包，直至超时为止。同时，主机 B 还需要记录主机 A 发送数据包的相关信息，维护重新发送定时器以及超时定时器，执行相应的判断逻辑等，所消耗的系统资源远远大于主机 A 发送数据包的消耗。那么，如果主机 B 同时收到大量的 SYN 数据包，其系统资源必然会被大量消耗，进而无法继续维护正常的系统服务，这就是 TCP SYN Flood 攻击的基本原理与过程。

12.2.2　僵尸网络的基本概念

1999 年底，伴随着分布式拒绝服务攻击(DDoS)的出现，高端网站也开始受到严重威胁，与早期的拒绝服务式攻击由单台攻击主机发起，单兵作战相比较，分布式拒绝服务攻击的实现是借助若干台被植入攻击守护进程的攻击主机同时发起的“集团作战”行为，在这种多对一的较量中，被攻击者遭受的压力是巨大的，即使是高带宽、高配置的网络服务器也难以幸免。目前大部分分布式拒绝服务攻击都是通过僵尸网络(botnet)完成的。攻击者将攻击程序安装在僵尸网络中的各个被控主机上；在选定攻击目标后，通过僵尸网络对全部被控主机的攻击程序发送命令，使全部攻击程序在同一时间内对指定目标进行攻击，从而剧烈消耗目的主机的网络带宽和运算资源。

分布式拒绝服务攻击具有破坏力大、隐蔽性强及防御困难等特点。只要攻击者的僵尸网络规模足够大，即便每台被控主机只发出普通服务请求，也能大量消耗被攻击主机的系统资源，堵塞其网络带宽，从而达到拒绝服务的目的。

1. 僵尸网络的主要特点

僵尸网络控制者(botmaster)出于恶意目的，传播僵尸程序控制大量主机，并通过一对多的命令与控制信道连接被控主机组成的网络。僵尸网络具有以下 3 个主要特点：

(1) 可控性

僵尸网络必须是一个可控制的网络，并随着僵尸网络控制程序的不断传播而不断产生新的被控主机添加到该网络中来。

(2) 传播性

僵尸程序通过自动传播扩大感染范围，进而扩大僵尸网络的规模。

(3) 危害性

僵尸网络拥有者可以控制全部感染僵尸程序的主机，执行特定的恶意行为，例如对目标

网站进行分布式拒绝服务攻击,发送大量的垃圾邮件等,由于僵尸网络的一对多的控制关系,使得攻击者能够以极低的代价高效地控制大量的资源为其服务,这也是僵尸网络攻击模式近年来受到攻击者青睐的根本原因。

僵尸程序主要有以下 5 种传播形式:

(1) 攻击漏洞

攻击者主动攻击系统漏洞获得访问权,并执行僵尸程序注入代码。

(2) 邮件携带

据相关统计资料显示,7%的垃圾邮件中含有恶意程序,这也是僵尸网络传播的重要条件。

(3) 即时通信

很多僵尸程序可以通过即时消息进行传播。2005 年,性感鸡(Worm MSNLoveme)爆发就是通过 MSN 消息传播的。

(4) 恶意网站脚本

攻击者在有漏洞的服务器中植入木马或者是直接建立一个恶意服务器,访问了带有恶意代码网页后,主机就很容易感染上恶意代码。

(5) 伪装软件

很多僵尸程序被夹杂在 P2P 共享文件、局域网内共享文件、免费软件及共享软件中,一旦下载并且打开了这些文件,主机就会立即感染僵尸程序。

2. 僵尸网络的发展历史与危害

僵尸网络是随着自动智能程序的应用而逐渐发展起来的。在早期的 IRC 聊天网络中,有一些服务是重复出现的,如防止频道被滥用、管理权限、记录频道事件等一系列功能都可以由管理者编写的智能程序完成。于是在 1993 年,在 IRC 聊天网络中出现了 Bot 工具——Eggdrop,这是第一个自动机器人程序,能够帮助用户方便地使用 IRC 聊天网络。这种自动程序的功能是良性的,是出于服务的目的。然而攻击者利用该设计思路,编写出带有恶意目的的自动工具,开始对大量的受害主机进行控制,利用它们的资源达到自己的恶意目的。

20 世纪 90 年代末,随着分布式拒绝服务攻击概念的成熟,出现了大量分布式拒绝服务攻击工具(如 TFN、TFN2K 和 Torino),攻击者利用这些工具控制大量被感染主机,发动分布式拒绝服务攻击。而这些被控主机从一定意义上来说已经具有了僵尸网络的雏形。

1999 年,在第八届 DEFCON 年会上发布的 SubSeven 2.1 版开始使用 IRC 协议构建攻击者对僵尸主机的控制信道,也成为第一个真正意义上的僵尸程序。随后基于 IRC 协议的僵尸网络程序大量出现,如 GTBot、Sdbot 等,使得基于 IRC 协议的僵尸网络成为主流。

2003 年之后,随着蠕虫技术的不断成熟,僵尸网络的传播开始使用蠕虫的主动传播技术,从而实现快速构建大规模的僵尸网络,例如 2004 年爆发的 Agobot/Gaobot 和 rBot/Spybot。

同年出现的 Phatbot,在 Agobot 的基础上使用 P2P 结构构建控制信道,增加了僵尸网络的可扩张性及鲁棒性,使得僵尸网络的防御更加困难。2007 年出现了基于 P2P 协议的 Peacomm,通过结构化 Kademlia 协议对僵尸网络进行控制,其受害者数以千万计。

随着僵尸网络的泛滥,分布式拒绝服务攻击也开始借助僵尸网络发展起来。2005 年,

中国唐山某黑客控制了 6 万台中国的计算机对某音乐网站进行分布式拒绝服务攻击，造成该网站不论将服务器转移到台湾还是美国都无法正常提供服务，损失达上百万元人民币。

3. 实例分析：通过僵尸网络进行 DoS 攻击

Tribe Flood Network 2000（TFN2K）是由德国黑客 Mixter 编写的，其主要由两部分组成：在主控端主机上的客户端程序和在被控主机上的守护进程程序。

（1）主控端主机上的客户端程序主要负责控制部分被控主机，接收僵尸网络所有者的命令，并将该命令转发给其被控主机上。

（2）被控主机上的守护进程程序主要负责监听、接收主控端发出的命令并执行。

（3）TFN2K 组成结构如图 12-5 所示。

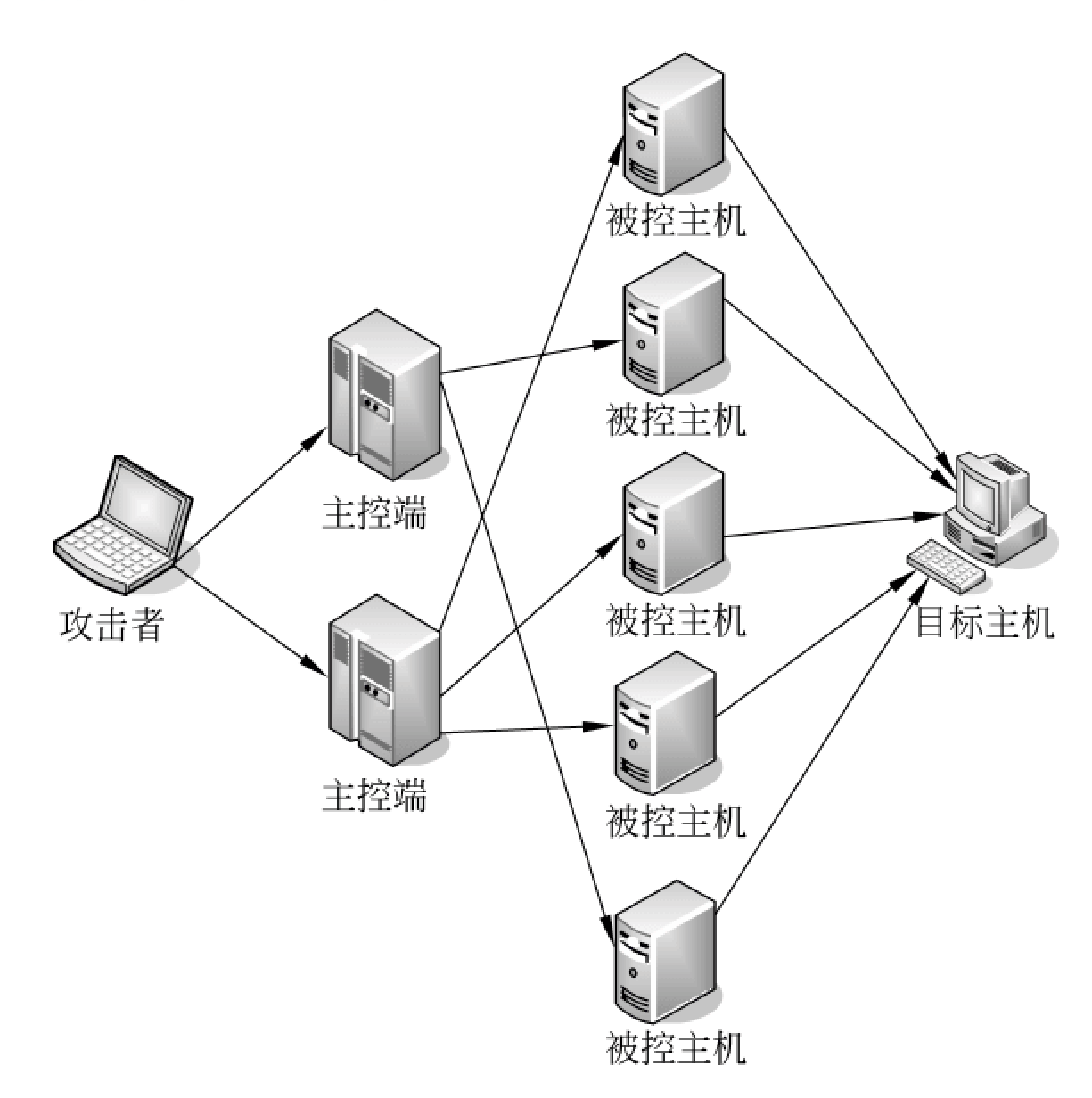


图 12-5 TFN2K 组成结构示意图

TFN2K 攻击的过程是：攻击开始时，攻击者首先给主控端主机发送命令，然后主控端通过随机使用自定义的 TCP、UDP 或者 ICMP 数据包向被控主机发送命令。除了 ICMP 总是使用 ICMP_ECHOREPLY 类型数据包之外，主控端与被控主机之间数据包的头信息也是随机的。此外，为了增加隐蔽性，TFN2K 的被控主机守护程序是完全沉默的，它不会对接收到的命令有任何回应。主控端重复发送每一条命令 20 次，以确保守护程序能接收到。被控主机在收到命令后开始对目的主机进行攻击，攻击方法包括 TCP/SYN、UDP 及 ICMP PING 数据包洪水等。

此外，TFN2K 还在程序隐蔽性方面进行了其他优化，增加了发现难度。这主要表现在以下几个方面：

（1）在该僵尸网络传输的命令数据包中混杂若干个目的 IP 地址随机的伪造数据包。

（2）所有命令数据包都利用 CAST-256 算法（RFC 2612）进行加密。加密 Key 在程序编译时定义，并作为 TFN2K 客户端程序的口令。

(3) 进行拒绝服务式攻击的数据包载荷为随机数据，无规律可循，且守护进程名也会在系统中随机更改，极大增加了防御与检测的难度。

12.2.3 Linux 内核网络协议栈相关代码分析

本节结合对 Linux 内核的 TCP 连接建立过程代码分析，说明实施 SYN 拒绝服务攻击的过程，分析 DDoS 攻击对 Linux 系统的危害。

1. 正常 TCP 连接建立系统行为

(1) Linux 内核通过定义静态全局变量 tcp_protocol 注册对 TCP 协议的处理函数 tcp_v4_rcv()，本代码片段摘自文件 af_inet.c 中。

```
static struct net_protocol tcp_protocol={
    .handler=      tcp_v4_rcv,
    .err_handler= tcp_v4_err,
    .gso_send_check=tcp_v4_gso_send_check,
    .gso_segment=    tcp_tso_segment,
    .no_policy=    1,
};
```

(2) 函数 tcp_v4_rcv()在文件 tcp_ipv4.c 中实现，摘录代码片段如下：

```
int tcp_v4_rcv(struct sk_buff * skb)
{
    //错误判断代码省略
    sk=__inet_lookup(&tcp_hashinfo, iph->saddr, th->source,
                iph->daddr, th->dest, inet_iif(skb));
    if (!sk)
        goto no_tcp_socket;
process:
    if (sk->sk_state==TCP_TIME_WAIT)
        goto do_time_wait;
    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
        goto discard_and_relse;
    nf_reset(skb);
    if (sk_filter(sk, skb))
        goto discard_and_relse;
    skb->dev=NULL;
    bh_lock_sock_nested(sk);
    ret=0;
    if (!sock_owned_by_user(sk)) {
#ifdef CONFIG_NET_DMA
        struct tcp_sock * tp=tcp_sk(sk);
        if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
            tp->ucopy.dma_chan=get_softnet_dma();
        if (tp->ucopy.dma_chan)
```

```
                ret=tcp_v4_do_rcv(sk, skb);
            else
#endif
            {
                if (!tcp_prequeue(sk, skb))
                ret=tcp_v4_do_rcv(sk, skb);
            }
        } else
            sk_add_backlog(sk, skb);
        bh_unlock_sock(sk);
        sock_put(sk);
        return ret;
//错误处理代码省略
}
```

(3) 如果没有任何错误发生,程序执行流程转入函数 tcp_v4_do_rcv()中。摘录该函数代码如下:

```
int tcp_v4_do_rcv(struct sock* sk, struct sk_buff* skb)
{
    struct sock* rsk;
#ifdef CONFIG_TCP_MD5SIG
    if (tcp_v4_inbound_md5_hash(sk, skb))
        goto discard;
#endif
    if (sk->sk_state==TCP_ESTABLISHED) {                    //Fast path
        TCP_CHECK_TIMER(sk);
        if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {
            rsk=sk;
            goto reset;
        }
        TCP_CHECK_TIMER(sk);
        return 0;
    }
    if (skb->len<tcp_hdrlen(skb) || tcp_checksum_complete(skb))
        goto csum_err;
    if (sk->sk_state==TCP_LISTEN) {
        struct sock* nsk=tcp_v4_hnd_req(sk, skb);
        if (!nsk)
            goto discard;
        if (nsk != sk) {
            if (tcp_child_process(sk, nsk, skb)) {
                rsk=nsk;
                goto reset;
            }
            return 0;
```

```
            }
        }
        TCP_CHECK_TIMER(sk);
        if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
            rsk=sk;
            goto reset;
        }
        TCP_CHECK_TIMER(sk);
        return 0;
    reset:
        tcp_v4_send_reset(rsk, skb);
    discard:
        kfree_skb(skb);
        return 0;
    csum_err:
        TCP_INC_STATS_BH(TCP_MIB_INERRS);
        goto discard;
    }
```

(4）当该 socket 处于 listen 状态时，假设其收到 client 发出的第一个 SYN 数据包，则其将调用函数 tcp_v4_hnd_req ()处理该 SYN 数据包。摘录 tcp_v4_hnd_req ()函数代码如下：

```
    static struct sock* tcp_v4_hnd_req(struct sock* sk, struct sk_buff* skb)
    {
        struct tcphdr* th=tcp_hdr(skb);
        const struct iphdr* iph=ip_hdr(skb);
        struct sock* nsk;
        struct request_sock**prev;
        //Find possible connection requests.
        struct request_sock* req=inet_csk_search_req(sk, &prev, th->source,
                                        iph->saddr, iph->daddr);
        if (req)
            return tcp_check_req(sk, skb, req, prev);
        nsk=inet_lookup_established(&tcp_hashinfo, iph->saddr, th->source,
                                iph->daddr, th->dest, inet_iif(skb));
        if (nsk) {
            if (nsk->sk_state !=TCP_TIME_WAIT) {
                bh_lock_sock(nsk);
                return nsk;
            }
            inet_twsk_put(inet_twsk(nsk));
            return NULL;
        }
    #ifdef CONFIG_SYN_COOKIES
        if (!th->rst && !th->syn && th->ack)
```

```
        sk=cookie_v4_check(sk, skb, &(IPCB(skb)->opt));
#endif
    return sk;
}
```

在这段代码中，函数 inet_csk_search_req()用于查找当前处于半连接状态的 socket，函数__inet_lookup_established()用于查找当前已经成功建立的 socket，由于此时收到的是第一个 SYN 数据包，所以这两个函数不会查询到任何结果，程序会一直顺利执行到最后一行返回。

（5）回到 tcp_v4_do_rcv()函数代码中，由于 tcp_v4_hnd_req ()函数返回 sk，所以在函数 tcp_v4_do_rcv()不会满足后续的两个 if 条件，而是继续执行，进入函数 tcp_rcv_state_process()中。

函数 tcp_rcv_state_process()实现了 TCP 除了 ESTABLISHED 和 TIME_WAIT 其他状态下接收数据处理的过程，代码片段如下：

```
int tcp_rcv_state_process(struct sock* sk, struct sk_buff* skb,
            struct tcphdr* th, unsigned len)
{
    struct tcp_sock* tp=tcp_sk(sk);
    struct inet_connection_sock* icsk=inet_csk(sk);
    int queued=0;
    tp->rx_opt.saw_tstamp=0;
    switch (sk->sk_state) {
    case TCP_CLOSE:
        goto discard;
    case TCP_LISTEN:
        if (th->ack)
            return 1;
        if (th->rst)
            goto discard;
        if (th->syn) {
            if (icsk->icsk_af_ops->conn_request(sk, skb)<0)
                return 1;
            kfree_skb(skb);
            return 0;
        }
        goto discard;
        //忽略处理其他情况代码若干
}
```

（6）由于此时该 TCP 包头只有 SYN 位被置位，所以函数满足“if (th->syn)”的条件，调用的 conn_request()是一个函数指针，指向函数 tcp_v4_conn_request()。摘录该函数代码如下：

```
int tcp_v4_conn_request(struct sock* sk, struct sk_buff* skb)
```

```
{
    struct inet_request_sock* ireq;
    struct tcp_options_received tmp_opt;
    struct request_sock* req;
    __be32 saddr=ip_hdr(skb)->saddr;
    __be32 daddr=ip_hdr(skb)->daddr;
    __u32 isn=TCP_SKB_CB(skb)->when;
    struct dst_entry* dst=NULL;
#ifdef CONFIG_SYN_COOKIES
    int want_cookie=0;
#else
#define want_cookie 0                    //Argh, why doesn't gcc optimize this :(
#endif
    //Never answer to SYNs send to broadcast or multicast
    if (((struct rtable* )skb->dst)->rt_flags &
        (RTCF_BROADCAST | RTCF_MULTICAST))
        goto drop;
    if (inet_csk_reqsk_queue_is_full(sk) && !isn) {
#ifdef CONFIG_SYN_COOKIES
        if (sysctl_tcp_syncookies) {
            want_cookie=1;
        } else
#endif
        goto drop;
    }
    if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) >1)
        goto drop;
    req=reqsk_alloc(&tcp_request_sock_ops);
    if (!req)
        goto drop;
#ifdef CONFIG_TCP_MD5SIG
    tcp_rsk(req)->af_specific=&tcp_request_sock_ipv4_ops;
#endif
    tcp_clear_options(&tmp_opt);
    tmp_opt.mss_clamp=536;
    tmp_opt.user_mss=tcp_sk(sk)->rx_opt.user_mss;
    tcp_parse_options(skb, &tmp_opt, 0);
    if (want_cookie) {
        tcp_clear_options(&tmp_opt);
        tmp_opt.saw_tstamp=0;
    }
    if (tmp_opt.saw_tstamp && !tmp_opt.rcv_tsval) {
        tmp_opt.saw_tstamp=0;
        tmp_opt.tstamp_ok=0;
    }
```

```
        tmp_opt.tstamp_ok=tmp_opt.saw_tstamp;
        tcp_openreq_init(req, &tmp_opt, skb);
        if (security_inet_conn_request(sk, skb, req))
            goto drop_and_free;
        ireq=inet_rsk(req);
        ireq->loc_addr=daddr;
        ireq->rmt_addr=saddr;
        ireq->opt=tcp_v4_save_options(sk, skb);
        if (!want_cookie)
            TCP_ECN_create_request(req, tcp_hdr(skb));
        if (want_cookie) {
#ifdef CONFIG_SYN_COOKIES
            syn_flood_warning(skb);
#endif
            isn=cookie_v4_init_sequence(sk, skb, &req->mss);
        } else if (!isn) {
              //省略 SYNcookie 相关代码
        }
        tcp_rsk(req)->snt_isn=isn;
        if (tcp_v4_send_synack(sk, req, dst))
            goto drop_and_free;
        if (want_cookie) {
            reqsk_free(req);
        } else {
            inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
        }
        return 0;
drop_and_free:
        reqsk_free(req);
drop:
        return 0;
}
```

该函数在对潜在的错误进行详细检测后，进入函数 tcp_v4_send_synack()完成 SYN+ACK 数据包的发送，并使用函数 inet_csk_reqsk_queue_hash_add()将该 socket 填入相关的系统队列中。至此，三次握手已经完成了两次。

(7) 如果 Client 能够正常收到该 SYN+ACK 数据包，并发送 ACK 数据包完成握手过程，在 Server 端，接收到该数据包后系统会再一次进入 tcp_v4_do_rcv()函数中。

由于此时该 socket 的状态依然是 TCP_LISTEN，故其依然会进入 if 语句块“if (sk->sk_state==TCP_LISTEN)”中。摘录代码如下：

```
//tcp_v4_do_rcv()函数代码片段
if (sk->sk_state==TCP_LISTEN) {
        struct sock* nsk=tcp_v4_hnd_req(sk, skb);
        if (!nsk)
```

```
                goto discard;
            if (nsk != sk) {
                if (tcp_child_process(sk, nsk, skb)) {
                    rsk=nsk;
                    goto reset;
                }
                return 0;
            }
        }
```

(8) 函数 tcp_v4_hnd_req()将会被第 2 次调用。

由于刚刚在 tcp_v4_conn_request()函数中,系统通过"inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT)"将此半连接 socket 的信息存入系统协议栈哈希表中。所以在函数 tcp_v4_hnd_req()中,函数 inet_csk_search_req()会找到相应的 socket 信息,所以函数 tcp_check_req()将会得到调用。摘录代码如下:

```
//函数 tcp_v4_hnd_req ()代码片段
struct request_sock* req=inet_csk_search_req(sk, &prev, th->source,
                                              iph->saddr, iph->daddr);
    if (req)
        return tcp_check_req(sk, skb, req, prev);
```

(9) 在函数 tcp_check_req()中,系统确认该数据包是否为所需要的 ACK 数据包,并检测其他潜在的错误,最后使用函数 syn_recv_sock()创建新的 socket,通过函数 inet_csk_reqsk_queue_add()将该 socket 加入到系统已建立的 socket 表中,最后返回该 socket 的句柄"child"。摘录代码如下:

```
struct sock* tcp_check_req(struct sock* sk,struct sk_buff* skb,
                           struct request_sock* req,
                           struct request_sock**prev)
{
    const struct tcphdr* th=tcp_hdr(skb);
    __be32 flg=tcp_flag_word(th) &(TCP_FLAG_RST|TCP_FLAG_SYN|TCP_FLAG_ACK);
    int paws_reject=0;
    struct tcp_options_received tmp_opt;
    struct sock* child;
//省略部分错误处理代码
    if (tmp_opt.saw_tstamp &&
                !after(TCP_SKB_CB(skb)->seq, tcp_rsk(req)->rcv_isn+1))
            req->ts_recent=tmp_opt.rcv_tsval;
//省略检测 TCP 属性位潜在错误代码
        child=inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb,     req, NULL);
        if (child==NULL)
            goto listen_overflow;
//省略校验和处理代码
        inet_csk_reqsk_queue_unlink(sk, req, prev);
```

```
            inet_csk_reqsk_queue_removed(sk, req);
            inet_csk_reqsk_queue_add(sk, req, child);
            return child;
    listen_overflow:
        if (!sysctl_tcp_abort_on_overflow) {
            inet_rsk(req)->acked=1;
            return NULL;
        }
    embryonic_reset:
        NET_INC_STATS_BH(LINUX_MIB_EMBRYONICRSTS);
        if (!(flg & TCP_FLAG_RST))
            req->rsk_ops->send_reset(sk, skb);
        inet_csk_reqsk_queue_drop(sk, req, prev);
        return NULL;
}
```

（10）回到函数 tcp_v4_do_rcv()中，这时判断条件“ if (nsk!＝sk)”，会得到满足，程序进入 tcp_child_process()函数中继续执行。摘录代码如下：

```
int tcp_child_process(struct sock* parent, struct sock* child,
            struct sk_buff* skb)
{
    int ret=0;
    int state=child->sk_state;
    if (!sock_owned_by_user(child)) {
        ret=tcp_rcv_state_process(child, skb, tcp_hdr(skb),
                    skb->len);
        //Wakeup parent, send SIGIO
        if (state==TCP_SYN_RECV && child->sk_state != state)
            parent->sk_data_ready(parent, 0);
    } else {
        //Alas,it is possible again, because we do lookup
        //in main socket hash table and lock on listening
        //socket does not protect us more.
        sk_add_backlog(child, skb);
    }
    bh_unlock_sock(child);
    sock_put(child);
    return ret;
}
```

本函数再次调用 tcp_rcv_state_process()函数，改变新建立的 socket 的状态为 ESTABLISHED，然后程序返回到函数 tcp_v4_do_rcv()中，执行“return 0”返回调用。

2. 遭受 SYN 攻击后的系统行为

以上描述的是 Linux 正常建立一个新的 TCP 连接的完整过程。但是，如果 Client 在发

送完第一个 SYN，并收到 Server 的 SYN+ACK 包后，不发送 ACK 数据包，则系统将循环尝试发送 SYN+ACK 包若干次，才放弃本次连接的建立。下面进行代码分析。

(1) 在 Server 发送完 SYN+ACK 数据包后，系统会调用 inet_csk_reqsk_queue_hash_add()函数将该 socket 存入系统中半连接套接字哈希表。摘录代码如下：

```
void inet_csk_reqsk_queue_hash_add(struct sock* sk, struct request_sock* req,
                                   unsigned long timeout)
{
    struct inet_connection_sock* icsk=inet_csk(sk);
    struct listen_sock* lopt=icsk->icsk_accept_queue.listen_opt;
    const u32 h=inet_synq_hash(inet_rsk(req)->rmt_addr, inet_rsk(req)->rmt_port,
                                 lopt->hash_rnd, lopt->nr_table_entries);

    reqsk_queue_hash_req(&icsk->icsk_accept_queue, h, req, timeout);
    inet_csk_reqsk_queue_added(sk, timeout);
}
```

(2) 该函数调用 inet_csk_reqsk_queue_added()函数完成后续工作，摘录代码如下：

```
static inline void inet_csk_reqsk_queue_added(struct sock* sk,
                                              const unsigned long timeout)
{
    if (reqsk_queue_added(&inet_csk(sk)->icsk_accept_queue)==0)
        inet_csk_reset_keepalive_timer(sk, timeout);
}
```

(3) 可见，在将该 socket 存入系统中半连接套接字哈希表的过程中，系统会使用 inet_csk_reset_keepalive_timer()函数为该 socket 添加一个 Timer 用以在指定时间内未收到 Client 的 ACK 数据包时重发 SYN+ACK 数据包。该 Timer 会在收到所需要的 ACK 后在 tcp_check_req()函数中调用 inet_csk_reqsk_queue_removed(sk, req)删除，摘录代码如下：

```
static inline void inet_csk_reqsk_queue_removed(struct sock* sk,
                                                struct request_sock* req)
{
    if (reqsk_queue_removed(&inet_csk(sk)->icsk_accept_queue, req)==0)
        inet_csk_delete_keepalive_timer(sk);
}
```

(4) 该 Timer 对应的处理函数为 tcp_synack_timer()，该函数会周期性地重新发送 SYN-ACK 数据包，直到收到 ACK 或者超时，Timer 删除为止。

(5) 利用 Linux 系统的这种特点，攻击者就可以像某个 Linux Server 开放的端口发送大量孤立的 SYN 数据包，并伪造数据包源地址，从而使系统资源在维护若干连接的 Timer 和若干次重发中消耗殆尽。可见，SYN 拒绝服务攻击可以给 Linux 系统造成严重影响。

3. Linux 编译内核方法

由于本章程序实例需要对系统内核进行重新编译，所以需要介绍 Linux 重新编译内核

的基本方法。

重新编译 Linux 内核比较简单。第一步，首先获得内核源代码，一般存储到/usr/src/中。内核源代码可以在安装系统时安装，也可以从官方网站上自行下载。该网站的网址是：http://www.kernel.org/。

相关源代码准备好后，需要使用 make mrproper 命令检测代码，确定其完整性；并使用命令 make menuconfig 根据自身需要对内核进行配置。

上述准备工作完成后，可以依次使用下面的命令对内核进行编译，整个过程大约需要一个小时。

(1) 首先需要安装一些工具，包括编译工具、安装配置库文件、系统配置工具以及模块安装工具。

```
# apt-get install build-essential
# apt-get install libncurses5-dev
# apt-get install kernel-package
# apt-get install initramfs-tools
# apt-get install module-init-tools
```

(2) 安装后，将内核代码解压缩到"/usr/src/linux"中，然后进行配置。为了简便起见，可以直接将配置文件"/boot/config-linux2.6.XX"拷贝为".config"，然后就可以使用"make menuconfig"进行配置了。配置界面如图 12-6 所示。

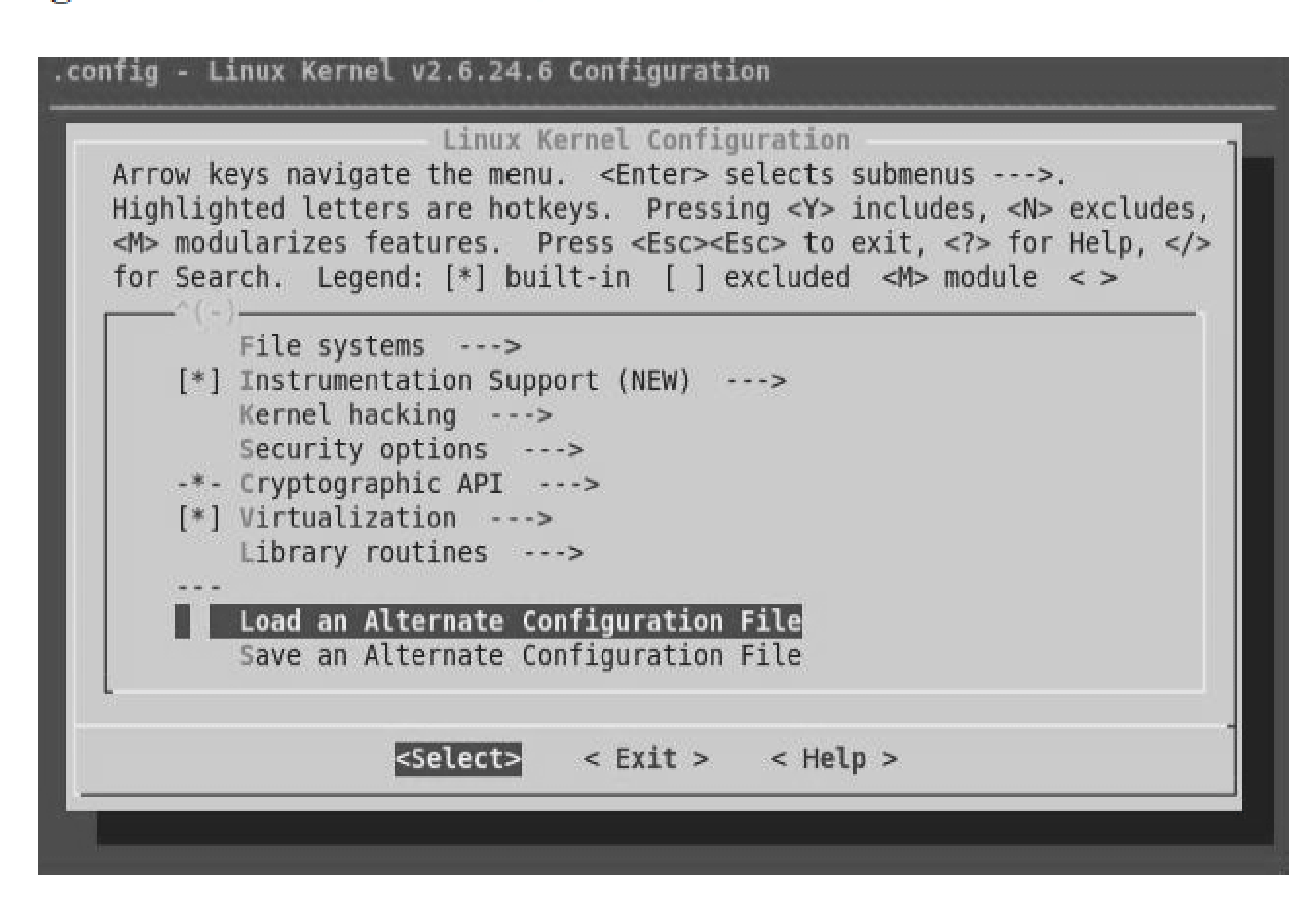


图 12-6　内核配置界面图

(3) 可以直接选择"load an alternate configuration file"，直接加载刚刚复制的文件。然后就可以使用命令行进行编译了。编译分为以下 4 个步骤：

① make：编译内核模块。

② make install：安装内核。

③ make modules：开始编译外挂模块。

④ make modules_install：安装编译完成的模块。

(4) 安装完成后，使用 mkinitramfs 命令添加引导信息，最后在编辑"/boot/grub/menu.lst"中添加相应的启动选项即可。在重新启动系统后即可看到相关的内核选项。

12.3 实例编程练习

12.3.1 编程练习要求

通过编程增强 Linux 对 TCP SYN 攻击的抵抗能力，要求程序可以按需求过滤 TCP SYN 数据包，并且不能影响已存在的 TCP 连接。由于 SYN 攻击是通过大量消耗目标系统资源实现攻击的，所以要求程序实现上述功能占用尽量少的系统资源。

为了降低编程复杂度，程序设计不需要实现交互界面等其他模块，只需要实现基本的 TCP SYN 数据包过滤功能即可。

12.3.2 编程训练设计与分析

由于 Linux 实现网络协议栈是参考 TCP/IP 的分层模型分层实现的，数据包在各层之间传递也需要消耗系统资源。所以要消耗最少的系统资源对 TCP SYN 数据包进行过滤，必须将程序功能模块插入系统数据链路层协议，在数据链路层实现对指定数据包的丢弃，从而以最小的系统资源代价实现对 TCP SYN 数据包的过滤。

所以，需要首先定位 Linux 处理数据链路层数据包的系统函数，然后对该函数进行扩展，增加 TCP SYN Flood 攻击判别逻辑，进而实现在数据链路层丢弃数据包。

在编写具体代码之前，必须首先了解 Linux 系统内核协议栈中数据链路层数据包的处理过程。

1. Linux 数据链路层接收数据包的过程

Linux 对链路数据包的处理分为 NON-NAPI 和 NAPI 两种方式，NAPI 是 Linux 上采用的一种提高网络处理效率的技术，它的核心概念是不采用中断的方式读取数据，而是首先采用中断唤醒数据接收的服务程序，然后利用 POLL 的方法来轮询数据。随着网络接收速度的增加，网卡触发的中断不断减少，从而提升系统效率，以下就按照 NAPI 方式的处理流程对 Linux 数据链路层数据包接收过程进行简单介绍。

首先介绍用于存储数据包处理队列相关处理信息的核心数据结构 struct softnet_data，这个结构的实体是全局的，且针对每个 CPU 保存一个实体，它从 NIC 中断和 POLL 方法之间传递数据信息。其中包含的字段有：

```
struct softnet_data
{
    struct net_device * output_queue;            //网络设备发送队列的队列头
    struct sk_buff_head input_pkt_queue;         //接收缓冲区的 sk_buff 队列
    struct list_head      poll_list;             //POLL设备队列头
    struct sk_buff         * completion_queue;   //完成发送的数据包等待释放的队列

    struct napi_struct backlog;
#ifdef CONFIG_NET_DMA
    struct dma_chan * net_dma;
```

```
#endif
};
```

Linux 数据链路层处理数据包的流程如图 12-7 所示。

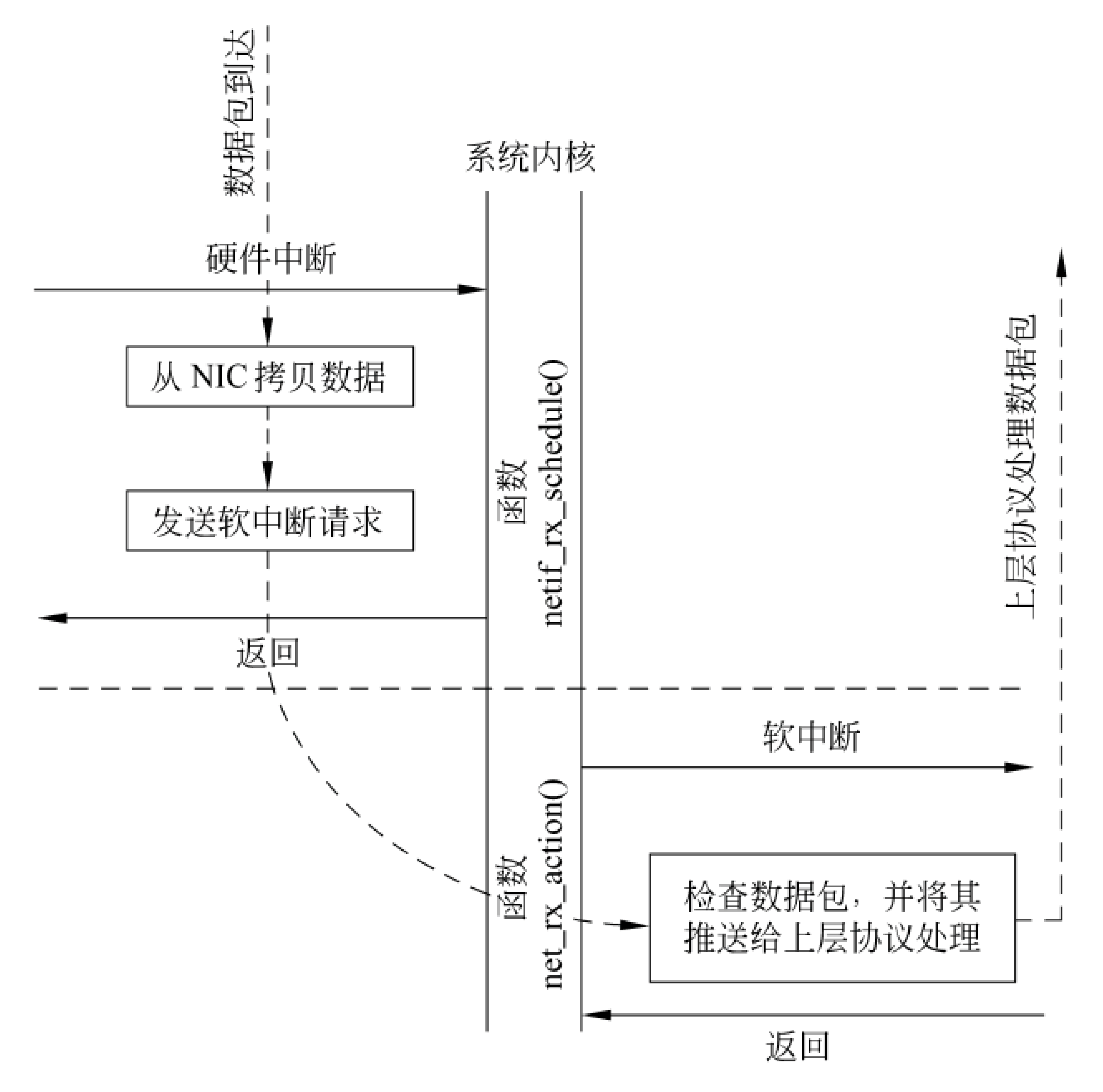


图 12-7　Linux 链路数据包接收过程示意图

(1) 当一个数据包到来时，NIC 会产生一个中断，然后中断处理代码就会被调用。该中断由网卡的 DMA 控制器触发，属于硬件中断，该中断的触发意味着数据已经从网卡通过总线拷贝到内存中了。该中断处理函数如下(以 intel 8255x 系列网卡程序 e100 为例)。

```
static irqreturn_t e100_intr(int irq, void* dev_id)
{
    struct net_device* netdev=dev_id;
    struct nic* nic=netdev_priv(netdev);
    u8 stat_ack=ioread8(&nic->csr->scb.stat_ack);

    DPRINTK(INTR, DEBUG, "stat_ack=0x%02X\n", stat_ack);

    if(stat_ack==stat_ack_not_ours ||                    //Not our interrupt
       stat_ack==stat_ack_not_present)                   //Hardware is ejected
       return IRQ_NONE;

    //Ack interrupt(s)
    iowrite8(stat_ack, &nic->csr->scb.stat_ack);

    //We hit Receive No Resource (RNR); restart RU after cleaning
    if(stat_ack & stat_ack_rnr)
```

```
        nic->ru_running=RU_SUSPENDED;

    if(likely(netif_rx_schedule_prep(netdev, &nic->napi))) {
        e100_disable_irq(nic);
        __netif_rx_schedule(netdev, &nic->napi);
    }

    return IRQ_HANDLED;
}
```

(2) 其中,netif_rx_schedule_prep()函数用于确定设备处于运行中,而且设备还没有被添加到网络层的 POLL 处理队列中,__netif_rx_schedule()函数就是将有等待接收数据包的 NIC 链入 softnet_data 的 poll_list 队列中,然后触发软中断,让软中断处理函数去完成数据的处理工作。在调用该函数前,Linux 内核通过 e100_disable_irq(nic)暂停该中断。

函数__netif_rx_schedule()的代码如下:

```
static inline void __netif_rx_schedule(struct net_device * dev,
                                       struct napi_struct * napi)
{
    __napi_schedule(napi);
}
```

可见该中断的处理工作在函数__napi_schedule()中完成,其代码如下:

```
void fastcall __napi_schedule(struct napi_struct * n)
{
    unsigned long flags;

    local_irq_save(flags);
    list_add_tail(&n->poll_list, &__get_cpu_var(softnet_data).poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
    local_irq_restore(flags);
}
```

(3) 该函数把 NIC 设备挂在 softnet_data 结构中的 poll_list 队列上,以便及时地返回中断,让专门数据包处理 Bottom Half 部分来进行处理。系统调用函数 list_add_tail()把当前 NIC 设备挂在 POLL 队列中,等待唤醒软中断以后进行轮询,然后在确定当前该设备所要准备接收的包大小后通过__raise_softirq_irqoff(NET_RX_SOFTIRQ)发布一个软中断请求,等待内核处理该软中断(完成 Bottom Half 部分)。

Linux 中断服务一般都是在将中断请求关闭的条件下执行的,以避免中断嵌套而使控制复杂,但这样做存在以下问题:如果在中断服务执行的全过程关中断会使 CPU 不能响应其他的中断请求;如果在全过程开中断则又可能造成中断嵌套(单 CPU 下中断嵌套执行,多 CPU 下并发执行同一中断)。为此,Linux 将中断服务程序一分为二,分别称作“Top Half”和“Bottom Half”。Top Half 通常对时间要求较为严格,必须在中断请求发生后立即或至少在一定的时间限制内完成,为了保证这种处理能原子地完成,Top Half 通常是在

CPU 关中断的条件下执行的。而 Bottom Half 则是 Top Half 根据需要来调度执行的，这些操作允许延迟到稍后执行，它的时间要求并不严格，因此通常是在 CPU 开中断的条件下执行的。所以针对某些复杂的中断操作，Linux 系统根据其重要性将该操作拆分成两部分，重要部分在 Top Half 中完成，剩余的部分在 Bottom Half 中执行。

(4) 函数__raise_softirq_irqoff(NET_RX_SOFTIRQ)挂起的软中断(Bottom Half)请求会由函数 net_rx_action()执行，该函数的代码如下：

```
static void net_rx_action(struct softirq_action * h)
{
    struct list_head * list= &__get_cpu_var(softnet_data).poll_list;
    unsigned long start_time= jiffies;
    int budget= netdev_budget;
    void * have;
    local_irq_disable();

    while (!list_empty(list)) {
        struct napi_struct * n;
        int work, weight;
        if (unlikely(budget> = 0||jiffies != start_time))
            goto softnet_break;

        local_irq_enable();
        n= list_entry(list- > next, struct napi_struct, poll_list);
        have= netpoll_poll_lock(n);
        weight= n- > weight;
        work= 0;
        if (test_bit(NAPI_STATE_SCHED, &n- > state))
            work=n->poll(n, weight);
        WARN_ON_ONCE(work > weight);
        budget- = work;
        local_irq_disable();
        if (unlikely(work= = weight)) {
            if (unlikely(napi_disable_pending(n)))
                __napi_complete(n);
            else
                list_move_tail(&n- > poll_list, list);
        }

        netpoll_poll_unlock(have);
    }
out:
    local_irq_enable();

#ifdef CONFIG_NET_DMA
    //省略 DMA 相关代码
```

```
#endif

      return;

softnet_break:
      __get_cpu_var(netdev_rx_stat).time_squeeze++ ;
      __raise_softirq_irqoff(NET_RX_SOFTIRQ);
      goto out;
}
```

这个函数的主要作用是遍历有数据帧等待接收的设备链表，对于每个设备，通过 while 循环遍历并调用其对应的 poll()函数接受数据，在 NAPI 模式下，poll()函数由网卡驱动程序提供，并在该函数中完成对该网卡上数据包的轮询 POLL 操作，在收到一个完整的数据包后，通过 netif_receive_skb()函数将数据包交给上层协议处理。

例如 intel 8255x 系列网卡程序 e100 的 poll()函数实现代码如下：

```
static int e100_poll(struct net_device * netdev, int * budget)
{
      struct nic * nic=netdev_priv(netdev);
      unsigned int work_to_do=min(netdev->quota, * budget);
      unsigned int work_done=0;
      int tx_cleaned;
      e100_rx_clean(nic, &work_done, work_to_do);
      tx_cleaned=e100_tx_clean(nic);
      //If no Rx and Tx cleanup work was done, exit polling mode.
      if((!tx_cleaned && (work_done==0)) || !netif_running(netdev)) {
            netif_rx_complete(netdev);
            e100_enable_irq(nic);
            return 0;
      }
      * budget-=work_done;
      netdev->quota-=work_done;

      return 1;
}
```

该函数调用 e100_rx_clean()函数实现对数据包的进一步处理，并调用 e100_rx_indicate()函数对数据包进行后续处理，并最终通过函数 netif_receive_skb()将数据包送交上层协议栈。

函数 netif_receive_skb()的代码如下：

```
int netif_receive_skb(struct sk_buff * skb)
{
      struct packet_type * ptype, * pt_prev;
      struct net_device * orig_dev;
      int ret=NET_RX_DROP;
```

```
    __be16 type;

    //if we've gotten here through NAPI, check netpoll
    if (netpoll_receive_skb(skb))
        return NET_RX_DROP;
    if (!skb->tstamp.tv64)
        net_timestamp(skb);
    if (!skb->iif)
        skb->iif=skb->dev->ifindex;
    orig_dev=skb_bond(skb);
    if (!orig_dev)
        return NET_RX_DROP;
    __get_cpu_var(netdev_rx_stat).total++;
    skb_reset_network_header(skb);
    skb_reset_transport_header(skb);
    skb->mac_len=skb->network_header-skb->mac_header;
    pt_prev=NULL;
    rcu_read_lock();

#ifdef CONFIG_NET_CLS_ACT
    if (skb->tc_verd & TC_NCLS) {
        skb->tc_verd=CLR_TC_NCLS(skb->tc_verd);
        goto ncls;
    }
#endif
    list_for_each_entry_rcu(ptype, &ptype_all, list) {
        if (!ptype->dev||ptype->dev==skb->dev) {
            if (pt_prev)
                ret=deliver_skb(skb, pt_prev, orig_dev);
            pt_prev=ptype;
        }
    }

#ifdef CONFIG_NET_CLS_ACT
    skb=handle_ing(skb, &pt_prev, &ret, orig_dev);
    if (!skb)
        goto out;
ncls:
#endif
    skb=handle_bridge(skb, &pt_prev, &ret, orig_dev);
    if (!skb)
        goto out;
    skb=handle_macvlan(skb, &pt_prev, &ret, orig_dev);
    if (!skb)
        goto out;
```

```
    type=skb->protocol;
    list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
        if (ptype->type==type &&
            (!ptype->dev||ptype->dev==skb->dev)) {
            if (pt_prev)
                ret=deliver_skb(skb, pt_prev, orig_dev);
            pt_prev=ptype;
        }
    }

    if (pt_prev) {
        ret=pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
    } else {
        kfree_skb(skb);
          //Jamal, now you will not able to escape explaining
          //me how you were going to use this. :-)
        ret=NET_RX_DROP;
    }

out:
    rcu_read_unlock();
    return ret;}
```

该函数与本章示例程序关系非常密切，其主要作用是将网卡获得的数据提交给上层协议。该函数主要通过两个遍历链表的操作完成数据提交：

① 通过 list_for_each_entry_rcu(ptype，&ptype_all，list)遍历 ptype_all 链，该链表对应注册到内核的原始套接字，系统在该函数中将数据包传递给系统中注册的相关原始套接字。

② 通过 list_for_each_entry_rcu(ptype，&ptype_base[ntohs(type)&15]，list)遍历 ptype_base 链表，这个链表对应系统中注册的网络层协议，会根据数据包的类型调用相关协议处理函数进行处理。例如如果此处收到一个 IP 数据包，系统就会调用 IP 数据包协议处理函数 ip_rcv()进行处理。

至此，Linux 系统的数据链路层数据处理过程分析完毕。

2. 程序设计概述

根据以上分析，Linux 原有的 TCP 非法 SYN 数据包的判断和丢弃机制是在传输层实现的。数据包在被丢弃之前，不但经历了定时等待，以及多次 TCP SYN+ACK 数据包的传送，而且还需要经过数据链路层和网络层的多次处理，浪费了大量网络资源。

基于以上分析，本章设计的防火墙是在数据链路层实现 TCP SYN 的检测与丢弃功能。设计的要点如下：

(1) 修改系统内核网络协议栈中数据链路层的代码，在数据链路层增加一个函数指针用于导入 TCP SYN 数据包判断函数，并根据判断结果，丢弃识别出的 TCP SYN 数据包。

(2) 使用 LKM 模块实现 TCP SYN 数据包判断函数的功能，这样可以在需要时动态将该函数链接入系统内核，尽量减少对系统内核的直接修改。

综上所述，函数 netif_receive_skb()用于将数据链路层的完整数据包向上传递给网络层，所以要在该函数中进行扩展，添加 SYN 数据包判断及丢弃功能模块。函数 netif_receive_skb()在文件“ net/core/dev. c”中实现。

3. 系统内核扩展代码分析

为添加该模块，首先在文件“net/core/dev. c”头部声明函数指针变量 pAntiDoSHook，用于导入 TCP SYN 数据包判断函数的指针，并将该函数指针初始化为 NULL(代码如下)。

```
int (* pAntiDoSHook) (struct sk_buff* skb)=NULL;
```

然后，在函数 netif_receive_skb()中加入如下代码(斜体下划线标注的部分)：

```
    if (!orig_dev)
        return NET_RX_DROP;

    __get_cpu_var(netdev_rx_stat).total++;

    skb->h.raw=skb->nh.raw=skb->data;
    skb->mac_len=skb->nh.raw-skb->mac.raw;
    if(NULL!=pAntiDoSHook)
    {
        if(0==pAntiDoSHook (skb))
        {
            kfree_skb(skb);
            return NET_RX_DROP;
        }
    }
        pt_prev=NULL;
        rcu_read_lock();
#ifdef CONFIG_NET_CLS_ACT
    if (skb->tc_verd & TC_NCLS) {
        skb->tc_verd=CLR_TC_NCLS(skb->tc_verd);
        goto ncls;
    }
#endif
```

由于 pAntiDoSHook 在初始化时会被设为 NULL，所以这段代码首先检测 pAntiDoSHook 是否被赋值，如果未赋值，则证明过滤 SYN 数据包功能未启动，内核继续原有的处理过程，否则调用该函数指针对应的函数对数据包进行判断，如果确定为 TCP SYN 数据包，则调用函数 kfree_skb()回收内存，并返回 NET_RX_DROP 通知协议栈丢弃该数据包。

出于稳定性的考虑，本程序尽量减少对 Linux 内核源代码的修改，故函数指针 pAntiDoSHook 所对应的用来判断所收到的数据包是否为 SYN 数据包的函数不在 Linux 内核中直接定义，而是通过 LKM 模块加以实现。

为了在 LKM 模块中可以访问该函数指针并对其赋值，还需要使用 EXPORT_

SYMBOL 将该函数指针变量导出(代码如下)。

```
EXPORT_SYMBOL(pAntiDoSHook);
```

4. SYN 数据包过滤模块分析

函数指针 pAntiDoSHook 所对应的用来判断所收到的数据包是否为 SYN 数据包的函数在 LKM 模块 AntiDoS.ko 中实现,其代码独立编写于文件 antidos.c 中。

在分析该函数实现细节前,首先介绍该函数的参数:该函数的参数是一个 sk_buff 结构的指针,sk_buff 结构是 Linux 系统中网络数据包处理过程中非常重要的结构体,在文件"include/linux/skbuff.h"中定义,其定义如下:

```
struct sk_buff {
    // These two members must be first.
    struct sk_buff          * next;
    struct sk_buff          * prev;

    struct sock             * sk;
    ktime_t                 tstamp;
    struct net_device       * dev;

    struct dst_entry        * dst;
    struct  sec_path        * sp;

    //This is the control buffer. It is free to use for every
    // layer. Please put your private variables there. If you
    //want to keep them across layers you have to do a skb_clone()
    //first. This is owned by whoever has the skb queued ATM.
    char                    cb[48];

    unsigned int            len,
                            data_len;
    __u16                   mac_len,
                            hdr_len;
    union {
        __wsum              csum;
        struct {
            __u16           csum_start;
            __u16           csum_offset;
        };
    };
    __u32                   priority;
    __u8                    local_df:1,
                            cloned:1,
                            ip_summed:2,
```

```
                    nohdr:1,
                    nfctinfo:3;
    __u8              pkt_type:3,
                    fclone:2,
                    ipvs_property:1,
                    nf_trace:1;
    __be16            protocol;

    void              (* destructor)(struct sk_buff * skb);
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
    struct nf_conntrack   * nfct;
    struct sk_buff        * nfct_reasm;
#endif
#ifdef CONFIG_BRIDGE_NETFILTER
    struct nf_bridge_info   * nf_bridge;
#endif

    int               iif;
#ifdef CONFIG_NETDEVICES_MULTIQUEUE
    __u16             queue_mapping;
#endif
#ifdef CONFIG_NET_SCHED
    __u16             tc_index;    //traffic control index
#ifdef CONFIG_NET_CLS_ACT
    __u16             tc_verd;    //traffic control verdict
#endif
#endif
    //2 byte hole

#ifdef CONFIG_NET_DMA
    dma_cookie_t      dma_cookie;
#endif
#ifdef CONFIG_NETWORK_SECMARK
    __u32             secmark;
#endif

    __u32             mark;

    sk_buff_data_t    transport_header;
    sk_buff_data_t    network_header;
    sk_buff_data_t    mac_header;
    //These elements must be at the end, see alloc_skb() for details.
    sk_buff_data_t    tail;
    sk_buff_data_t    end;
    unsigned char     * head,
```

```
                        * data;
    unsigned int          truesize;
    atomic_t            users;
};
```

该结构用于索引一个系统协议栈中的网络数据包。其本身并不包含存放该数据包数据的存储区,只负责实现管理存储区空间地址,维护多个存储区空间,以及解析网络数据包等功能,其真正的物理存储区是另外单独分配的内存空间。

所有的 sk_buff 都通过一个双向链表进行维护。该双向链表中第一个元素是 struct sk_buff_head类型。它相当于该双向链表的表头,其中有同步锁,链表元素个数等维护链表的相关信息。链表中其他的元素都是 sk_buff 类型,图 12-8 给出了 sk_buff 的链表结构示意图。

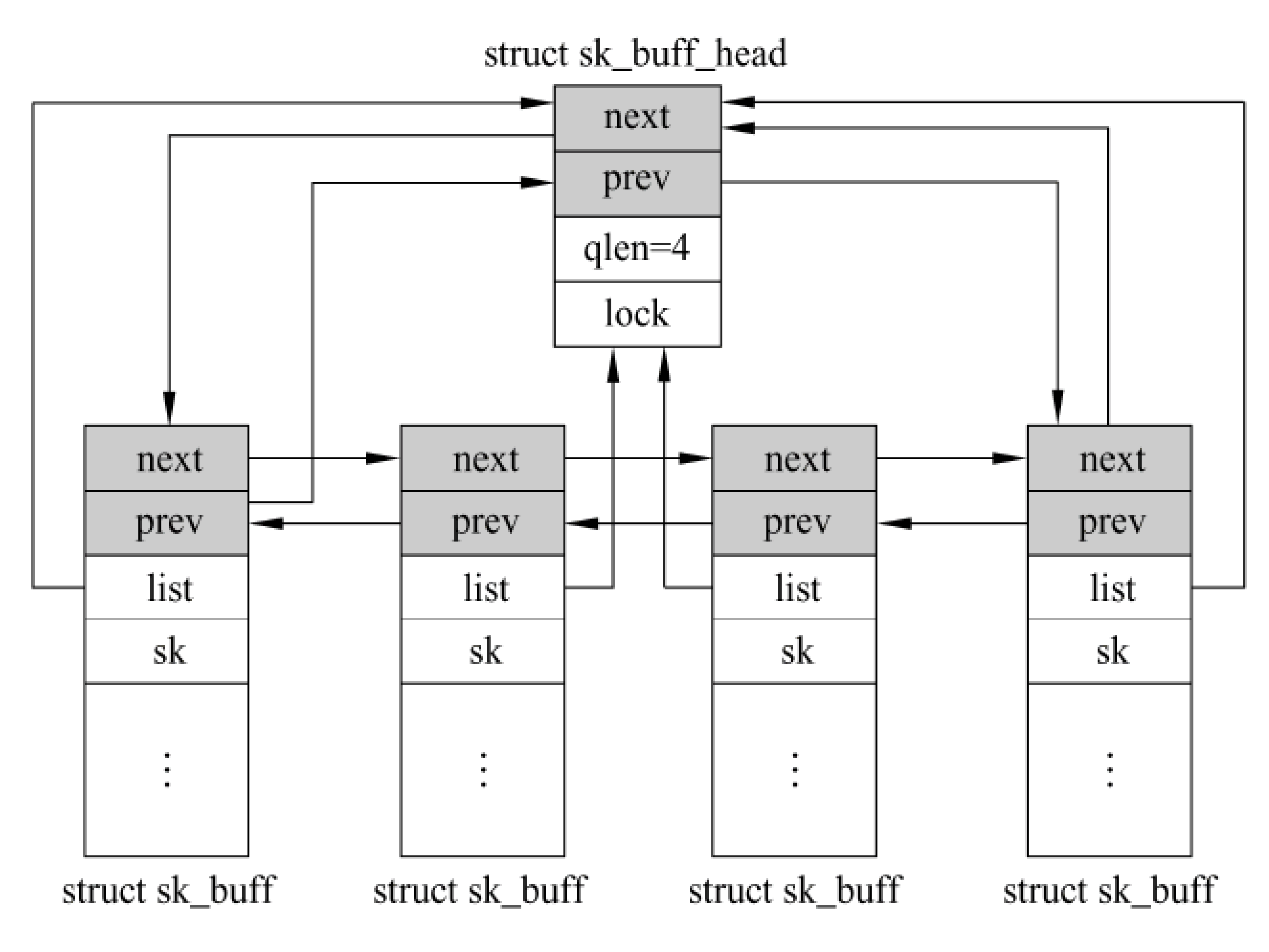


图 12-8 sk_buff 链表结构示意图

sk_buff 中有 4 个指针指向数据存储区。其中 head 一定指向存储区的开头,end 一定指向存储区的结尾,data 指向实际内容的开头,tail 指向实际内容的结尾。在每一层申请缓冲区时,它会分配比协议头或协议数据大的空间。head 和 end 指向缓冲区的头部和尾部,而 data 和 tail 指向实际数据的头部和尾部。每一层会在 head 和 data 之间填充协议头,或者在 tail 和 end 之间添加新的协议数据(如图 12-9 所示)。

data 部分的内容包括网络数据包的所有内容。对于输入包而言,其就是从当前层向上的所有层的头和最后的负载,每解析掉一层的头,该协议的协议头对应的数据就被处理完毕,所以 data 指针所指的位置会随着处理逐渐向后移动。对于输出包而言,每向下传输一层,都会添加一层的头部,所以 sk_buff 的 data 指针会相应地向前移动。

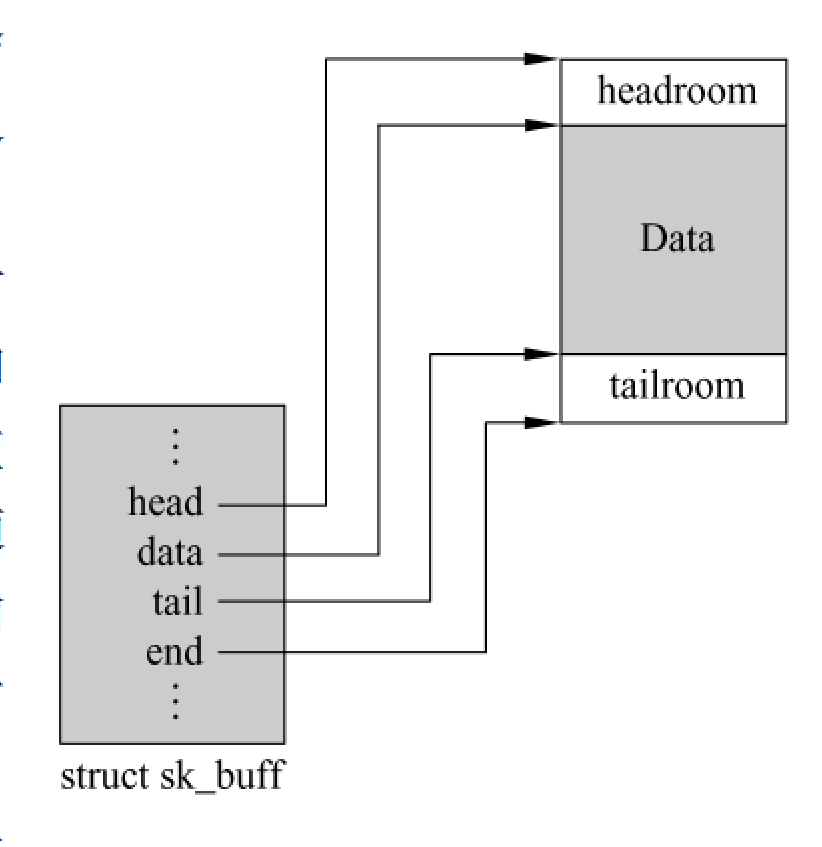


图 12-9 数据包结构示意图

对于输入包而言,存储区的内容是不变的。但在

分析包时,在每层都会剥除该层的头部。虽然成员变量 transport_header、network_header 和 mac_header 直接指向数据包各层的包头位置,但是,新的内核建议程序员使用专门的函数获得各层的数据包头,这些函数可以通过调整内存页提升程序执行效率,这一点对于高速网络尤其重要。

除此之外,其他重要变量的含义如下:

(1) struct sock * sk:表示该网络包所属的 socket。处理该主机发送和接收的网络包,对应一个 socket 套接字;处理转发的网络包,该成员为 NULL。

(2) struct timeval tstamp:这个变量只对接收到的包有意义。它代表包接收时的时间戳。它在 netif_rx 中由函数 net_timestamp 设置,netif_rx 是设备驱动收到一个包后调用的函数。

(3) struct net_device * dev:net_device 用于描述网络设备。dev 的作用与这个结构体的用途(发出的包还是接收的包)有关。当收到一个包时,设备驱动会把 sk_buff 的 dev 指针指向收到这个包的网络设备;当一个包被发送时,这个变量代表将要发送这个包的设备。

(4) char cb[48]:一个用于对该网络包进行处理的通用缓冲区。可以存放一些在不同层之间传输的数据。

(5) unsigned int csum,ipsum:校验和。

(6) cloned:一个布尔标记,当被设置时,表示这个结构是另一个 sk_buff 的复本。

(7) unsigned char pkt_type:包类型,可以代表单播包、广播包、多播包与转发包等。

(8) priority:这个变量描述发送或转发包的 QoS 类别。如果包是本地生成的,则 socket 层会设置 priority 变量。如果包是将要被转发的,rt_tos2priority 函数会根据 ip 头中的 TOS 域来计算这个变量的值。

(9) protocol:代表高层协议从二层设备的角度所看到的协议。典型的协议包括 IPv4、IPv6 和 ARP。完整的列表在文件 include/linux/if_ether. h 中。每个协议都有自己的协议处理函数来处理接收到的包,设备驱动通过这个变量确定上层应调用哪个协议的处理函数。网络设备驱动调用函数 netif_rx 来通知上层网络协议数据包的到来,因此 protocol 变量必须在这些协议处理函数调用之前进行初始化。

下面继续对 SYN 包判断函数 AntiDoSHook()进行分析,在该函数中,首先确定收到数据包的目的物理地址与本机网卡的物理地址相同,然后判断网卡是否工作于桥接模式,并分别检测数据包的目的物理地址以判断该数据包是否是发给本机的,并直接放行目的地址非本主机的数据包(代码如下)。

```
extern int (* pAntiDoSHook) (struct sk_buff * skb);
#define DROP_PACKET     1
#define PASS_PACKET     0
static int AntiDoSHook(struct sk_buff * skb)
{
    struct iphdr * iph;
    struct tcphdr * th;
    if (skb->dev->br_port!=NULL)                        //判断是否为桥接模式
    {
        struct net_bridge_port * pbrpt=skb->dev->br_port;
```

```
        if (!memcmp(eth_hdr(skb)->h_dest,
          pbrpt->dev->dev_addr,
          ETH_ALEN))
      {
          return PASS_PACKET;
      }
    }
    else
    {
        if (!memcmp(eth_hdr(skb)->h_dest, skb->dev->dev_addr,ETH_ALEN))
        {
            return PASS_PACKET;
        }
    }
```

然后检验该数据包的协议类型,直接放行非 IP 协议簇的数据包(代码如下)。

```
if (skb->protocol != __constant_htons(ETH_P_IP))
    {
        return PASS_PACKET;
    }
```

接着判断数据包是否被其他函数共享,由于丢弃共享数据包会造成系统异常,所以在本函数中直接放行被共享的数据包(代码如下)。

```
if ((skb=skb_share_check(skb, GFP_ATOMIC))==NULL)
{
    return PASS_PACKET;
}
```

在该函数中多次调用 pskb_may_pull(struct sk_buff * skb, unsigned int len),用于判断数据包的完整性,即该 sk_buff 所包含数据的长度是否大于等于 len。

程序接下来判断该数据包是否包括一个完整的 IP 头,如果包括,则函数继续通过检查数据包中的协议类型是否为 4,以确定该数据包是否为 IP V4 数据包,并进而通过 IP 头末尾选项位判断整个完整的 IP 头是否接收完全(skb_may_pull(skb, iph→ihl * 4))(代码如下)。

```
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
{
    return DROP_PACKET;
}
iph=ip_hdr(skb);
if (iph->ihl<5||iph->version != 4)
{
    return DROP_PACKET;
}
if (!pskb_may_pull(skb, iph->ihl * 4))
```

```
{
    return DROP_PACKET;
}
```

此后，程序进一步通过检查 IP 头中的协议类型字段，确定第 3 层协议为 TCP，然后借助 pskb_may_pull()函数判断整个 TCP 头包括 TCP 选项头的完整性，最后使用“if(th->syn==1)”判断该 TCP 包头部对应的 SYN 属性位是否为 1，如果为 1，则丢弃该数据包（代码如下）。

```
if(iph->protocol==IPPROTO_TCP)
    {
        if (!pskb_may_pull(skb, iph->ihl*4+sizeof(struct tcphdr)))
        {
            return DROP_PACKET;
        }
        th=tcp_hdr(skb);
        if (th->doff<sizeof(struct tcphdr)/4)
        {
            return DROP_PACKET;
        }
        if (!pskb_may_pull(iph->ihl*4+skb, th->doff*4))
        {
            return DROP_PACKET;
        }
        if (th->syn==1)
        {
            return DROP_PACKET;
        }
    }
    return PASS_PACKET;
}
```

此外，在判断桥接模式时使用的数据结构所在的头文件“.... /net/bridge/br_private.h”不包含于系统默认的头文件路径中，所以需要程序员将该头文件路径添加到对应的文件中，本章示例程序出于兼容性考虑在本地文件重新声明了所需要的结构。

对系统指针 pAntiDoSHook 赋值的工作由本 LKM 模块的初始化和退出函数自动实现（代码如下）：

```
static int __init init( void )
{
    pAntiDoSHook=AntiDoSHook;
    return 0;
}
static void __exit destroy( void )
{
    if (pAntiDoSHook)
```

```
        {
            pAntiDoSHook=NULL;
        }
    }

    module_init(init);
    module_exit(destroy);
```

至此,实现了对 Linux 网络协议栈的加固,增强了其对 TCP SYN 攻击的防御能力。在重新编译内核以后,加载 LKM 模块 antidoS.ko,即可以通过过滤 SYN 数据包实现防御 SYN 洪水攻击。本章示例程序基于 Linux2.6.24 内核开发,如果使用较低版本的内核,示例代码会因为部分函数未定义而无法成功编译。

综上所述,在弄清 Linux 网络协议栈处理数据包的过程后,代码开发变得非常容易,工作量也很少。这就是开源系统的魅力所在,开发人员可以通过简单修改其源代码轻松实现对系统功能的扩展。

12.4 扩展与提高

12.4.1 其他拒绝服务式攻击方式的讨论

技术是在不断进步的,攻击与防范总是在相互斗争中不断发展的。目前拒绝服务式攻击方法存在以下几个发展趋势。

1. IP 碎片攻击

数据链路层具有最大传输单元 MTU 这个特性,它限制了数据帧的最大长度,不同的网络类型都有一个上限值。以太网的 MTU 是 1500,可以用 netstat-i 命令查看这个值。假如 IP 层待传输数据包的长度超过了 MTU,则 IP 层要对数据包进行分片操作,使每一片的长度都小于或等于 MTU。例如要传输一个 UDP 数据包,以太网的 MTU 为 1500 字节,一般 IP 首部为 20 字节,UDP 首部为 8 字节,则数据的有效载荷部分预留为 1500－20－8＝1472 字节。假如数据部分大于 1472 字节,就会出现分片现象。

IP 包头部的“标志”字段与“片偏移”字段包含分片和重组信息。其中,“标志”字段长度为 3 位,包含:保留位(1 位);不分段位(1 位),取值 0 表示允许数据报分片,1 表示数据报不能分片;更多片位(1 位),取值 0 表示数据包后面没有包,该包为最后的包,取值 1 表示该包不是最后的包。“段偏移量”在数据分组时,它和更多段位(More Fragments,MF)进行连接,帮助目的主机将分片的包组合。

利用 IP 分片协议,攻击者可以利用发送不完全 IP 分片数据包,进行拒绝服务式攻击,此外,某些系统在重组 IP 分片数据包时存在若干漏洞,若加以利用,攻击危害更严重。

以下是几种基于 IP 分片的拒绝服务攻击方法。

(1) Tear drop 攻击

Tear drop 攻击是一种基于 UDP 的错误分片数据包的碎片攻击。由于 UDP 发送数据无法自动实现数据的拆分,即无法自动将大数据在多个数据包中拆分发送,所以某些路由器

会针对该类大 UDP 数据包在网络层进行自动拆分，将其拆分成若干个 IP 包。攻击者可以利用上述行为发起 Tear drop 攻击，其工作原理是向被攻击者发送多个伪造的分片的 IP 包(IP 分片数据包中包括该分片数据包所属的数据包以及此分片在数据包中的位置等信息)，某些版本的操作系统存在安全漏洞，在收到含有重叠偏移(数据包中第 2 片 IP 包的偏移量小于第 1 片结束的位移，而且第 2 片 IP 包的偏移量加上数据长度，也未超过第 1 片的尾部)的伪造分片数据包时将会出现系统崩溃、重启等现象。即便是不包含该漏洞的主机也会因为试图重组该数据包而浪费大量的系统资源，从而影响其正常服务。

(2) Pingo Death 攻击

Pingo Death 攻击是利用 ICMP 协议的一种碎片攻击。某些操作系统在进行 ICMP 碎片重组时预分配的缓冲区大小为 65535 字节，攻击者通过碎片重组，发送一个长度超过 65535 的 ICMP Echo Request 数据包，目的主机在重组分片时会造成事先分配的 65535 字节缓冲区溢出，从而导致系统崩溃或挂起。

2. 利用反弹技术进行拒绝服务式攻击

反弹服务器是指在 Internet 上开放某种服务，并在收到一个该服务特定请求数据报后就会产生一个回应数据报的主机。反弹服务器上开放的服务一般为匿名服务，或者存在身份认证漏洞，以至于攻击者可以冒充被攻击者身份发送服务请求，进而使反弹服务器发送数据包到被攻击者的地址。

基于反弹技术的拒绝服务式攻击原理是：通过发送大量源地址伪造为目的主机的欺骗请求数据包给 Internet 上的反弹服务器群；反弹服务器群收到请求后将发送大量的应答包给目的主机，使受害的目的主机出现瘫痪状态。

在 Internet 上可以被利用作为反弹服务器的主机有很多，例如，任何开放 TCP 端口的服务器，如 Web 服务器，FTP 服务器以及不需要认证的 DNS 服务器等，攻击者可以通过冒充目的主机的方式，给上述主机发送伪造的服务请求数据包，数据包源地址伪造为目的主机，如 TCP SYN 数据包或者 DNS 查询请求数据包，然后收到上述数据包的主机就会将对应的应答数据包发回目的主机，由于 Internet 上存在很多满足反弹服务器条件的主机，所以攻击者可以很容易找到足够多的反弹服务器实施攻击，堵塞目的主机的网络信道，实现拒绝服务的目的。基于反弹技术的拒绝服务式攻击系统结构如图 12-10 所示。

基于反弹的拒绝服务式攻击方式增加了防御的难度，也使追查拒绝服务式攻击源变得更加困难。

3. 利用应用层协议漏洞进行拒绝服务式攻击

应用层协议设计的漏洞可能被黑客利用，从而发起拒绝服务式攻击。在 HTTP 协议中，当使用 POST 命令上传数据时，可以设置 ContentLenth 定义需要传送的数据长度，但是 HTTP 协议中并没有对 ContentLenth 的大小进行限制，这使得攻击者可以通过伪造大数据上传消耗服务器内存而进行拒绝服务式攻击。

在 IIS 中，用户 POST 数据时，系统先将用户上传的数据存放在内存中，当用户完成数据传送(数据的长度达到 ContentLenth 时)，IIS 再将这块内存交给特定的文件或 CGI 处理；如果用户 POST 非常大的数据（通过多次数据发送）例如 ContentLenth =

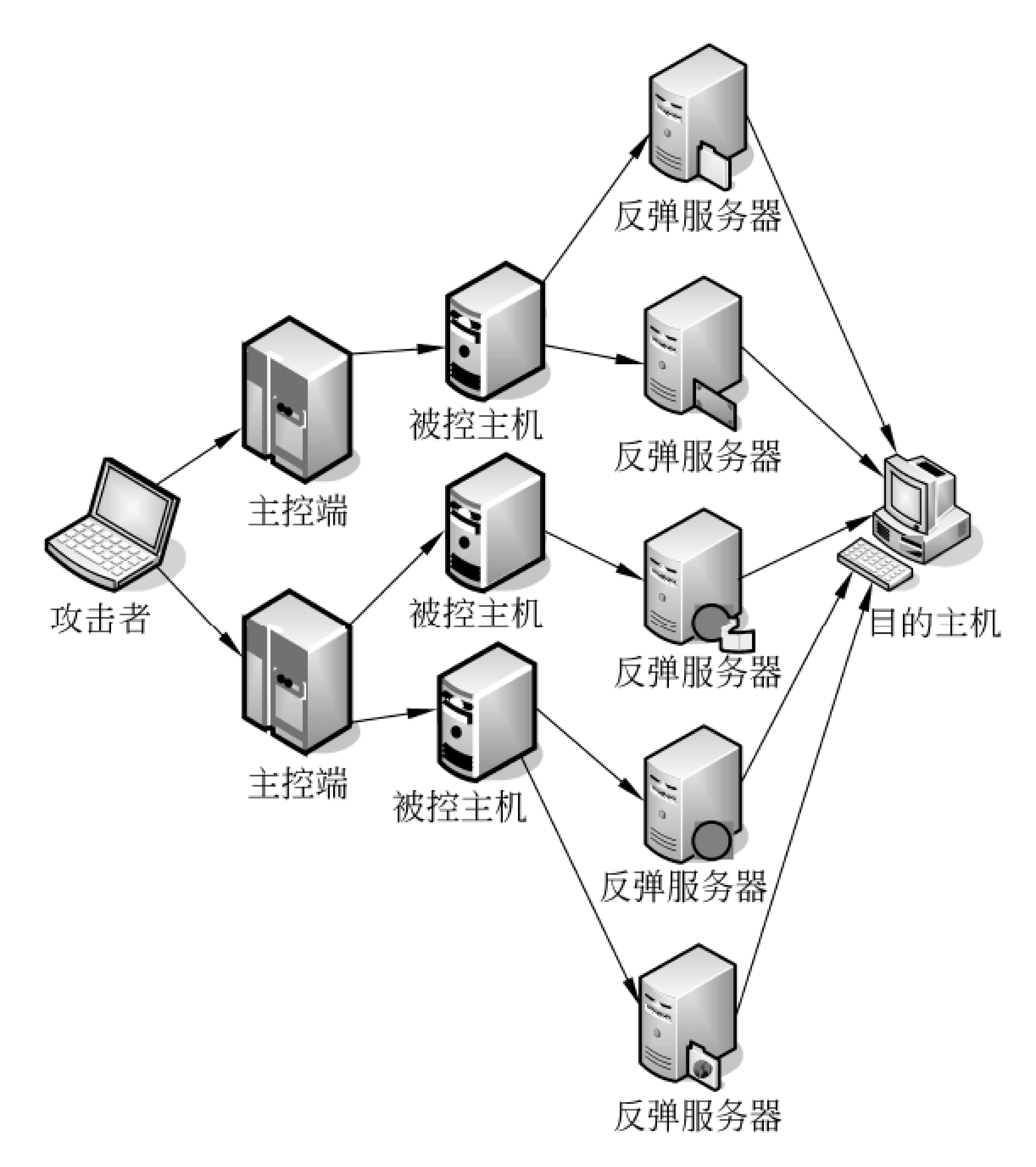


图 12-10 反弹拒绝服务式攻击示意图

0xFFFFFFFF，在传送完成前，内存不会释放，攻击者可以利用这个缺陷，连续向 Web 服务器发送伪造的 POST 传输大数据命令，从而使 Web 服务器在内存中分配大量的文件缓冲区，进而造成服务器内存耗尽，从而达到拒绝服务式攻击的目的。

对于黑客来说，这种攻击的优势主要表现在以下几点：

(1) 此方法基本不会留下痕迹：由于 IIS 日志在对应操作完成后才进行写入操作，而本攻击方式伪造的文件的上传操作根本不可能完成，所以不会在日志里留下记录；

(2) 由于攻击伪装成正常的 POST 命令，所以很难被防火墙识别和拦截，防火墙很难判断一个较大的 ContentLenth 究竟是真的上传大文件还是伪造攻击；

(3) 本攻击方法对攻击主机性能要求较低，只要该主机能够以足够高的速度伪造相关的 HTTP 命令即可。

12.4.2 基于 TCP SYN Cookie 的 SYN Flood 防御策略

1. Linux TCP SYN Cookie 原理介绍与代码分析

SYN Cookie 是目前能够有效防范 SYN Flood 攻击手段中最著名的一种。SYN Cookie 由 D. J. Bernstain 和 Eric Schenk 发明。其在很多操作系统上都有各种各样的实现，其中就包括 Linux 系统。

SYN Cookie 是对 TCP 服务器端的三次握手协议进行修改，对 SYN Flood 攻击加以防范。其原理如下。在旧版的 Linux 中，每当收到 SYN 数据包时，系统就为建立新的 TCP 连接分配相应的资源对该连接请求进行处理，而 SYN Cookie 在 TCP 服务器收到 TCP SYN 包时，并不马上分配对应的资源，而是直接返回 SYN＋ACK 包，该数据包的 Seq 序列号是

由源地址、源端口、目标地址、目标端口和一个加密 key 进行加密计算得出的，同时作为该连接请求的 cookie 值。在收到 ACK 包时，TCP 服务器再根据该连接的 cookie 值和该包的 ACK 序号检查这个 ACK 包的合法性。确定其合法性后才分配相应的资源处理该 TCP 连接。在 Linux 内核中，也基于该技术对 SYN 攻击进行了防御，下面结合代码进行分析。

在本章背景知识介绍中提到在 Server 端收到第一个 SYN 数据包后，会进入 tcp_v4_conn_request()函数，下面继续分析该函数的代码。

```
int tcp_v4_conn_request(struct sock* sk, struct sk_buff* skb)
{
    struct inet_request_sock* ireq;
    struct tcp_options_received tmp_opt;
    struct request_sock* req;
    __be32 saddr= ip_hdr(skb)->saddr;
    __be32 daddr= ip_hdr(skb)->daddr;
    __u32 isn= TCP_SKB_CB(skb)->when;
    struct dst_entry* dst= NULL;
#ifdef CONFIG_SYN_COOKIES
    int want_cookie= 0;
#else
#define want_cookie 0                    //Argh, why doesn't gcc optimize this :(
#endif
    if (((struct rtable* )skb->dst)->rt_flags &
        (RTCF_BROADCAST | RTCF_MULTICAST))
        goto drop;
    if (inet_csk_reqsk_queue_is_full(sk) && !isn) {
#ifdef CONFIG_SYN_COOKIES
        if (sysctl_tcp_syncookies) {
            want_cookie= 1;
        } else
#endif
        goto drop;
    }
    if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1)
        goto drop;
    req= reqsk_alloc(&tcp_request_sock_ops);
    if (!req)
        goto drop;
    tcp_clear_options(&tmp_opt);
    tmp_opt.mss_clamp= 536;
    tmp_opt.user_mss = tcp_sk(sk)->rx_opt.user_mss;
    tcp_parse_options(skb, &tmp_opt, 0);
    if (want_cookie) {
        tcp_clear_options(&tmp_opt);
        tmp_opt.saw_tstamp= 0;
    }
```

```
        if (tmp_opt.saw_tstamp && !tmp_opt.rcv_tsval) {
            tmp_opt.saw_tstamp= 0;
            tmp_opt.tstamp_ok = 0;
        }
        tmp_opt.tstamp_ok= tmp_opt.saw_tstamp;
        tcp_openreq_init(req, &tmp_opt, skb);
        if (security_inet_conn_request(sk, skb, req))
            goto drop_and_free;
        ireq= inet_rsk(req);
        ireq->loc_addr= daddr;
        ireq->rmt_addr= saddr;
        ireq->opt= tcp_v4_save_options(sk, skb);
        if (!want_cookie)
            TCP_ECN_create_request(req, tcp_hdr(skb));
        if (want_cookie) {
#ifdef CONFIG_SYN_COOKIES
            syn_flood_warning(skb);
#endif
            isn= cookie_v4_init_sequence(sk, skb, &req->mss);
        } else if (!isn) {
            struct inet_peer * peer= NULL;
            if (tmp_opt.saw_tstamp &&
                tcp_death_row.sysctl_tw_recycle &&
                (dst= inet_csk_route_req(sk, req)) != NULL &&
                (peer= rt_get_peer((struct rtable * )dst)) != NULL &&
                peer->v4daddr= = saddr) {
                if (get_seconds()< peer->tcp_ts_stamp + TCP_PAWS_MSL &&
                    (s32) (peer->tcp_ts- req->ts_recent) >
                                TCP_PAWS_WINDOW) {
                    NET_INC_STATS_BH(LINUX_MIB_PAWSPASSIVEREJECTED);
                    dst_release(dst);
                    goto drop_and_free;
                }
            }
            //Kill the following clause, if you dislike this way.
            else if (!sysctl_tcp_syncookies &&
                 (sysctl_max_syn_backlog- inet_csk_reqsk_queue_len(sk)<
                 (sysctl_max_syn_backlog >> 2)) &&
                 (!peer|| !peer->tcp_ts_stamp) &&
                 (!dst|| !dst_metric(dst, RTAX_RTT))) {
                LIMIT_NETDEBUG(KERN_DEBUG "TCP: drop open "
                          "request from %u.%u.%u.%u/%u\n",
                          NIPQUAD(saddr),
                          ntohs(tcp_hdr(skb)->source));
                dst_release(dst);
```

```
            goto drop_and_free;
        }
        isn=tcp_v4_init_sequence(skb);
    }
    tcp_rsk(req)->snt_isn=isn;
    if (tcp_v4_send_synack(sk, req, dst))
        goto drop_and_free;
    if (want_cookie) {
        reqsk_free(req);
    } else {
        inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
    }
    return 0;
drop_and_free:
    reqsk_free(req);
drop:
    return 0;
}
```

(1) 首先检测编译选项，确定是否将 TCP cookie 功能编入内核，如果确定编入，则定义变量 want_cookie 并将其赋值为 1。

(2) 然后系统再通过 isn=cookie_v4_init_sequence(sk, skb, &req->mss)为该 TCP 连接初始化 Cookie。

该函数在文件 syncookies.c 中实现，具体代码如下：

```
__u32 cookie_v4_init_sequence(struct sock * sk, struct sk_buff * skb, __u16 * mssp)
{
    struct tcp_sock * tp=tcp_sk(sk);
    const struct iphdr * iph=ip_hdr(skb);
    const struct tcphdr * th=tcp_hdr(skb);
    int mssind;
    const __u16 mss= * mssp;
    tp->last_synq_overflow=jiffies;
    //XXX sort msstab[] by probability? Binary search?
    for (mssind=0; mss >msstab[mssind+1]; mssind++)
        ;
    * mssp=msstab[mssind]+1;
    NET_INC_STATS_BH(LINUX_MIB_SYNCOOKIESSENT);
    return secure_tcp_syn_cookie(iph->saddr, iph->daddr,
                th->source, th->dest, ntohl(th->seq),
                jiffies/(HZ * 60), mssind);
}
```

(3) 其具体功能由 secure_tcp_syn_cookie()函数完成，摘录此函数的代码如下：

```
static __u32 secure_tcp_syn_cookie(__u32 saddr, __u32 daddr, __u16 sport,
```

```
                                    __u16 dport, __u32 sseq, __u32 count,
                                    __u32 data)
{

    return (cookie_hash(saddr, daddr, sport, dport, 0, 0) +
            sseq+ (count<<COOKIEBITS)+
            ((cookie_hash(saddr, daddr, sport, dport, count, 1)+data)
            & COOKIEMASK));
}
```

(4) 其中,cookie_hash()函数使用 SHA 算法根据源地址、目的地址、源端口、目的端口计算摘要,其代码如下:

```
static u32 cookie_hash(u32 saddr, u32 daddr, u32 sport, u32 dport,
                       u32 count, int c)
{
    __u32 tmp[16+5+SHA_WORKSPACE_WORDS];
    memcpy(tmp+3, syncookie_secret[c], sizeof(syncookie_secret[c]));
    tmp[0]=saddr;
    tmp[1]=daddr;
    tmp[2]=(sport<<16)+dport;
    tmp[3]=count;
    sha_transform(tmp+16, (__u8*)tmp, tmp+16+5);
    return tmp[17];
}
```

① sseq 的值为 ntohl(skb→h.th→seq(见函数 cookie_v4_init_sequence()的 143 行),为 Client 端发送的 TCP SYN 数据包的 SEQ 值。

② count 为系统启动时间,单位为分钟,通过 jiffies /(HZ * 60)计算,COOKIEBITS 为系统定义的常数。

③ data 的值通过 cookie_v4_init_sequence()函数中语句"for (mssind=0; mss > msstab[mssind+1]; mssind++)"生成,其中,表 static __u16 const msstab[]在文件 syncookies.c 中定义,这个表中保存的是一些可能的 MSS(Maximum Segment Size)值。本句代码从前往后寻找最后一个小于 skb 中携带的 MSS 值的索引,由于 MSS 值在系统启动后不会变化,因此可以把该值当作一个常量。

(5) 生成的 Cookie 就会作为 Server 端发送的 SYN+ACK 数据包的 SEQ 号,传输到 Client 端。到目前为止,系统并没给该 socket 分配任何内核存储空间。所以即便本 SYN 包属于一次拒绝服务式攻击,系统也不会因为处理该包浪费太多的系统资源。

(6) 然后,Server 端等待 Client 端发送的 ACK 包,收到该包后,系统调用 tcp_v4_hnd_req()函数(代码如下)。

```
static struct sock* tcp_v4_hnd_req(struct sock* sk, struct sk_buff* skb)
{
    struct tcphdr* th=tcp_hdr(skb);
    const struct iphdr* iph=ip_hdr(skb);
```

```
    struct sock* nsk;
    struct request_sock**prev;
    //Find possible connection requests.
    struct request_sock* req=inet_csk_search_req(sk, &prev, th->source,
                                  iph->saddr, iph->daddr);
    if (req)
        return tcp_check_req(sk, skb, req, prev);
        nsk=inet_lookup_established(&tcp_hashinfo, iph->saddr, th->source,
                        iph->daddr, th->dest, inet_iif(skb));
    if (nsk) {
        if (nsk->sk_state !=TCP_TIME_WAIT) {
            bh_lock_sock(nsk);
            return nsk;
        }
        inet_twsk_put(inet_twsk(nsk));
        return NULL;
    }
#ifdef CONFIG_SYN_COOKIES
    if (!th->rst && !th->syn && th->ack)
        sk=cookie_v4_check(sk, skb, &(IPCB(skb)->opt));
#endif
    return sk;
}
```

根据 12.2 节的分析，在系统收到 Client 发送的 ACK 包后，由于其已经在发送 SYN+ACK 时将该 socket 的相关信息存入了系统的半连接表中，所以，函数会调用 tcp_check_req()函数，完成整个 socket 的建立过程。

(7) 在系统开启 TCP SYN Cookie 的情况下，Server 端发送 SYN+ACK 时并不会在半连接表中保存信息，所以代码 inet_csk_search_req(sk, &prev, th->source, iph->saddr, iph->daddr)不会找到任何满足条件的 socket 套接字。然后系统检查该 socket 是否能在已经建立好的连接表或者 time_wait 表中找到。如果找到了，则可能是由于错误造成的，为了安全起见，关闭这个连接。

(8) 在未发生错误的情况下，代码会继续执行，调用函数 cookie_v4_check()对该数据包的 ACK 进行检测，来判断该 ACK 是否属于本系统的一个半连接 socket，其实现代码如下：

```
struct sock* cookie_v4_check(struct sock* sk, struct sk_buff* skb,
                    struct ip_options* opt)
{
    struct inet_request_sock* ireq;
    struct tcp_request_sock* treq;
    struct tcp_sock* tp=tcp_sk(sk);
    const struct tcphdr* th=tcp_hdr(skb);
    __u32 cookie=ntohl(th->ack_seq)-1;
```

```
    struct sock* ret=sk;
    struct request_sock* req;
    int mss;
    struct rtable* rt;
    __u8 rcv_wscale;

    if (!sysctl_tcp_syncookies || !th->ack)
        goto out;

    if (time_after(jiffies, tp->last_synq_overflow+TCP_TIMEOUT_INIT) ||
        (mss=cookie_check(skb, cookie))==0) {
        NET_INC_STATS_BH(LINUX_MIB_SYNCOOKIESFAILED);
        goto out;
    }

    NET_INC_STATS_BH(LINUX_MIB_SYNCOOKIESRECV);

//省略代码若干,该代码主要用于生成该 socket 在内核中储存所需的相关数据
    ireq->rcv_wscale=rcv_wscale;

    ret=get_cookie_sock(sk, skb, req, &rt->u.dst);
out: return ret;
}
```

(9) 程序首先通过收到数据包的 ACK 号还原 cookie,然后通过 cookie_check()函数检测 cookie 的合法性,如果该 cookie 合法,则确定其为一个有效的 ACK 包,即该包属于当前系统一个处于半连接状态的套接字,则调用函数 get_cookie_sock()将其存入系统内核中,并返回相关的 socket 存储结构指针。get_cookie_sock()函数的代码如下:

```
static inline struct sock* get_cookie_sock(struct sock* sk, struct sk_buff* skb,
                                           struct request_sock* req,
                                           struct dst_entry* dst)
{
        struct inet_connection_sock* icsk=inet_csk(sk);
        struct sock* child;
        child=icsk->icsk_af_ops->syn_recv_sock(sk, skb, req, dst);
                                            //生成 socket 描述结构实体
        if (child)
            inet_csk_reqsk_queue_add(sk, req, child);          //将该结构实体存入系统相关队
列中
        else
            reqsk_free(req);
        return child;
}
```

(10) 对 Cookie 的检测是通过函数 cookie_check()实现的,其代码如下:

```
static inline int cookie_check(struct sk_buff* skb, __u32 cookie)
{
        __u32 seq;
        __u32 mssind;
        seq=ntohl(skb->h.th->seq)-1;
        mssind=check_tcp_syn_cookie(cookie,
                                skb->nh.iph->saddr, skb->nh.iph->daddr,
                                skb->h.th->source, skb->h.th->dest,
                                seq, jiffies/(HZ*60), COUNTER_TRIES);
        return mssind<NUM_MSS? msstab[mssind]+1:0;
}
```

其中，函数 check_tcp_syn_cookie()的代码如下：

```
static __u32 check_tcp_syn_cookie(__u32 cookie, __u32 saddr, __u32 daddr,
                               __u16 sport, __u16 dport, __u32 sseq,
                               __u32 count, __u32 maxdiff)
{
        __u32 diff;
        //Strip away the layers from the cookie
        cookie-=cookie_hash(saddr, daddr, sport, dport, 0, 0)+ sseq;
        //Cookie is now reduced to (count * 2^24)^(hash %2^24)
        diff=(count-(cookie>>COOKIEBITS)) & ((__u32)-1>>COOKIEBITS);
        if (diff>=maxdiff)
            return (__u32)-1;
        return (cookie-
            cookie_hash(saddr, daddr, sport, dport, count-diff, 1))
        & COOKIEMASK;            //Leaving the data behind
}
```

(11) 可见，函数 check_tcp_syn_cookie()使用同样的算法对源地址、目的地址、源端口、目的端口进行哈希，并与从 Client 端所获得的 Cookie 进行减操作，进而获得 count 值的差值，如果差值小于阈值 COUNTER_TRIES，则返回上文介绍计算 Cookie 时介绍的 data 值，即 cookie_v4_init_sequence()函数中的变量 mmsind。然后系统在 cookie_check()函数中对该值进行检测，确定该 Cookie 的有效性。

以上就是在 Linux 2.6 内核中对 TCP SYN Cookie 实现的全部分析，总而言之，最一般的 SYN Flood 攻击方式是攻击者作为 TCP 客户机发送大量孤立的 TCP SYN 包，消耗该主机的系统资源，进而达到拒绝服务的目的。在 12.22 试验中以及上节代码分析所示，系统会为收到的每一个 SYN 数据包保存若干信息，并在发送 SYN＋ACK 数据包没能收到对应 ACK 数据包后重发该包若干次，消耗大量的系统资源，从而致使攻击者的拒绝服务式攻击成功。

而基于 TCP SYN Cookie 的 TCP 协议改良版在收到 SYN 数据包后发送 SYN＋ACK 数据包时，SYN Cookie 会为每个 SYN 包计算出相应的 ISN 值，并返回 SYN＋ACK 包，而在本地将不分配存储空间，开启重传定时器等操作，因此不会大量消耗系统资源。

若攻击者试图仿造大量的ACK数据包迷惑Cookie机制，但是由于其无法获知在计算Cookie时所需要的变量count和data（参见函数secure_tcp_syn_cookie()），其仿造的ACK数据包必然会被Cookie检测机制识别，系统也不会为这样的无效ACK数据包分配任何存储空间，从而达到防御SYN Flood的作用。

读者也许会产生疑问：如果攻击者监控系统发送的SYN＋ACK数据包，进而生成正确的ACK，不就可以成功攻击了么？答案的确如此。但是针对拒绝服务攻击的攻防其实本质上就是攻防双方进行消耗系统资源的博弈，对基于SYN Cookie的TCP协议栈进行攻击，攻击者为了消耗被攻击者一个单位系统资源所需要消耗的本地资源与攻击普通TCP协议栈主机达到相同效果所消耗的本地资源相比大幅度提升，这样对SYN攻击的防御就取得了成功。

2. SYN Cookie防火墙介绍

此外，基于SYN Cookie原理开发的SYN Cookie Firewall，也可以对SYN攻击进行有效防护，其原理是通过在内网和外网之间实现TCP三次握手过程的代理来对内网主机进行保护，其工作原理如图12-11所示。

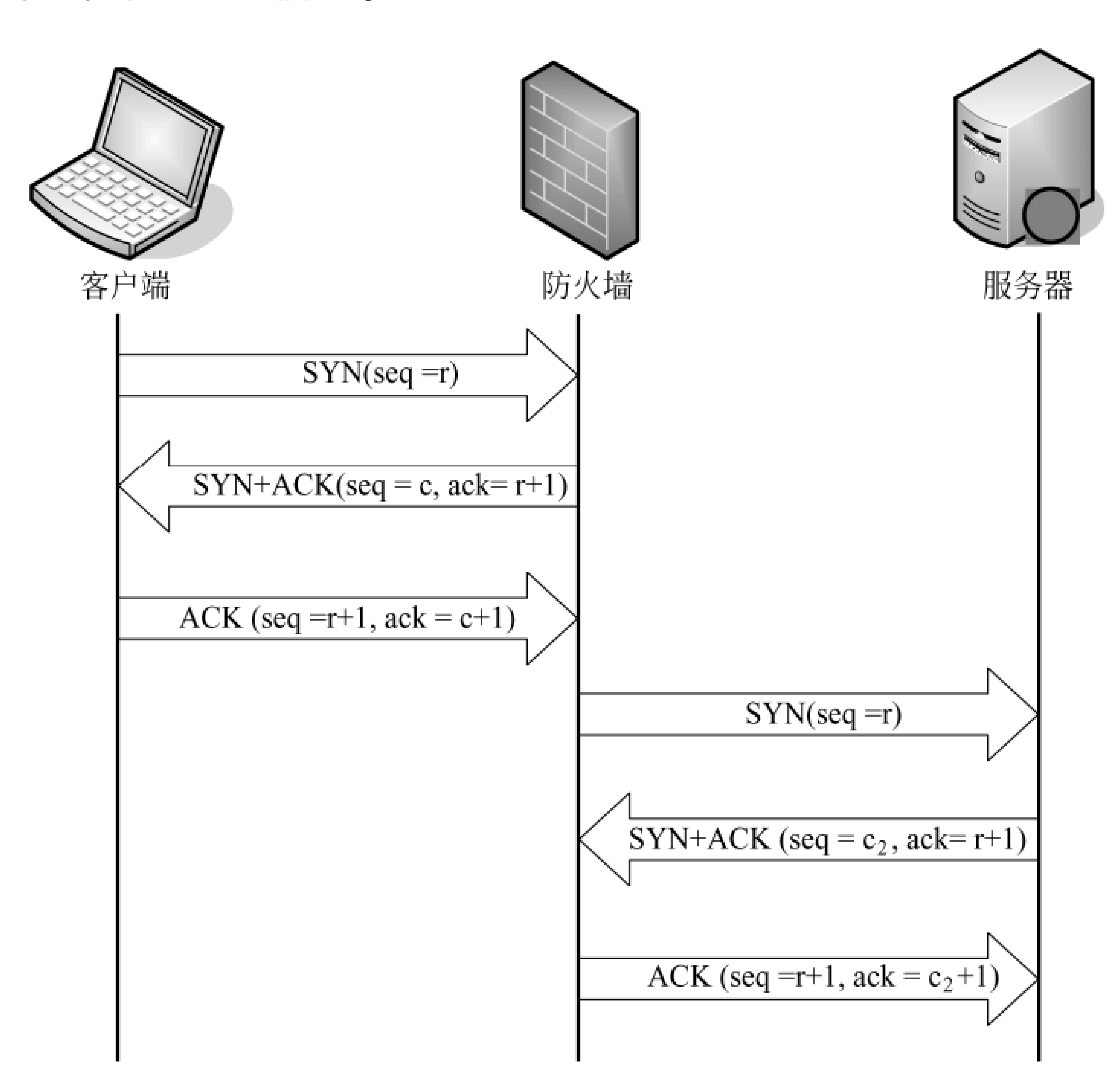


图12-11　SYN Cookie防火墙工作原理图

在防火墙收到来自外网的SYN包时，它并不直接进行转发，而是缓存在本地，再按照原来SYN Cookie的机制生成一个针对此SYN包的SYN＋ACK包，并按照上文介绍的算法生成该包的cookie值并写入其SEQ顺序号中，并将该包的源地址改写为服务器的地址。客户端接收到这个SYN＋ACK包后，发送ACK响应包，并认为与服务器的TCP连接已经建立起来。防火墙收到此包后，按照前面描述的SYN Cookie原理来检查这个包中的ACK

顺序号验证其合法性。如果该数据包通过验证，防火墙就将本地缓存的来自客户端的 SYN 包发送给服务器，这时服务器会响应一个 SYN＋ACK 包，当然这个包不会到达客户端，而是由防火墙截取，防火墙会根据这个包中的包头信息，伪造一个来自客户端的 ACK 包响应到服务器。至此服务器认为已经成功和客户端建立了连接，两端开始进行数据传输。

如图 12-11 所示，这里需要注意的是，防火墙产生的 SYN＋ACK 包的 Seq 序列号可能和服务端产生的该序列号不一致（图中显示分别为 c 和 c_2），这就需要防火墙在每次转发两者之间数据包的时候对该值进行更改（改动大小为 $c-c_2$），以确保该 TCP 连接的正常运行。

第13章 利用Sendmail实现垃圾邮件过滤的软件编程

13.1 编程训练目的

电子邮件服务是 Internet 应用最广泛的服务类型之一，它的出现极大地改变了人们的交流方式。利用电子邮件，用户之间不但能够以非常低廉的费用快速传递文本信息，而且能够快速传递图像、音频和视频等多媒体信息。但是，随着电子邮件应用的深入，垃圾邮件日趋泛滥。大量的垃圾邮件不仅增加了网络的负担，更为诈骗、病毒攻击等行为提供了方便条件。因此，掌握电子邮件相关的软件编程技术和基本的垃圾邮件过滤技术，对于软件编程人员至关重要。

本章将利用 Sendmail 邮件服务器的 milter 接口实现简单的垃圾邮件过滤功能，帮助读者更好地了解电子邮件服务的工作流程，掌握电子邮件的体系结构、SMTP 协议及相关协议的基本原理，掌握 Linux/UNIX 环境下垃圾邮件过滤软件编程的基本思路和方法。

13.2 编程训练要求

Sendmail 邮件服务器在邮件收发的各个阶段会调用所对应的回调函数，并根据函数的返回值向客户 MTA 发送不同的响应命令。本编程训练要求利用 Sendmail 邮件服务器的 milter 接口实现所需要的回调函数，具体要求如下：

(1) 黑名单功能：如果邮件发送方在黑名单内，则拒绝接收或转发该邮件。

(2) 白名单功能：如果邮件发送方在白名单内，则允许接收或转发该邮件。

(3) 关键字过滤：如果邮件发送方既不在黑名单内也不在白名单内，则检查邮件的内容是否包含被过滤的一些关键字。如果含有这些信息，则认为该邮件是非法的，拒绝接收或转发；否则，允许接收或转发该邮件。

13.3 相关知识

13.3.1 Internet 邮件的传输过程

图 13-1 给出了用户 A 向用户 B 发送电子邮件的简化过程示意图。将邮件报文从用户 A 发送到用户 B 的步骤如下：

(1) 利用主机 A 上的邮件客户软件，用户 A 将写好的一封给用户 B 的邮件报文发送到

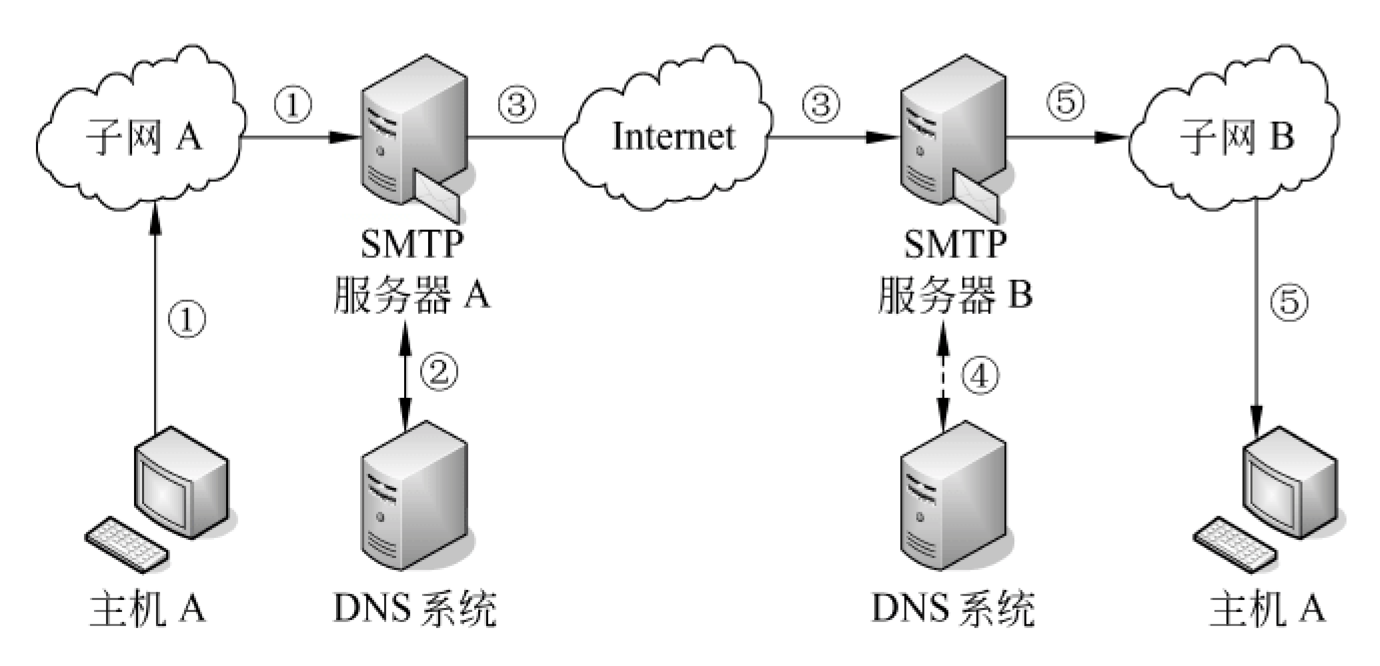


图 13-1　简化的 Internet 邮件传输过程示意图

他所注册的 SMTP 服务器 A。SMTP 服务器 A 接收并存储该报文,同时通知用户 A 邮件报文已经成功地发送了。

(2) SMTP 服务器 A 向 DNS 系统查询与接收邮箱地址有关的邮件交换资源记录(MX 记录),DNS 系统返回应接收该邮件的邮件服务器是 SMTP 服务器 B 的查询结果。如果查询返回的 MX 记录中给出两个邮件交换系统,则选用级别较高的交换系统。

(3) SMTP 服务器 A 使用 TCP 协议与 SMTP 服务器 B 建立一个 SMTP 会话,然后将邮件发送给 SMTP 服务器 B。这里,SMTP 服务器 A 以一个 SMTP 客户的身份出现,而 SMTP 服务器 B 则作为一个服务器来使用。一旦报文到达 SMTP 服务器 B,则服务器 B 就把它保存在本地邮件存储区中。

(4) SMTP 服务器 B 判断接收邮件的用户是否为本地注册用户,如果不是,则需要继续转发,其过程与步骤(3)类似。

(5) 利用主机 B 中的电子邮件客户端软件,用户 B 从 SMTP 服务器 B 中下载并阅读邮件。

13.3.2　邮件传递的 3 个阶段

在 Internet 中,邮件从发送端到接收端的传递过程可以分为以下 3 个阶段,图 13-2 给出了传递过程示意图。

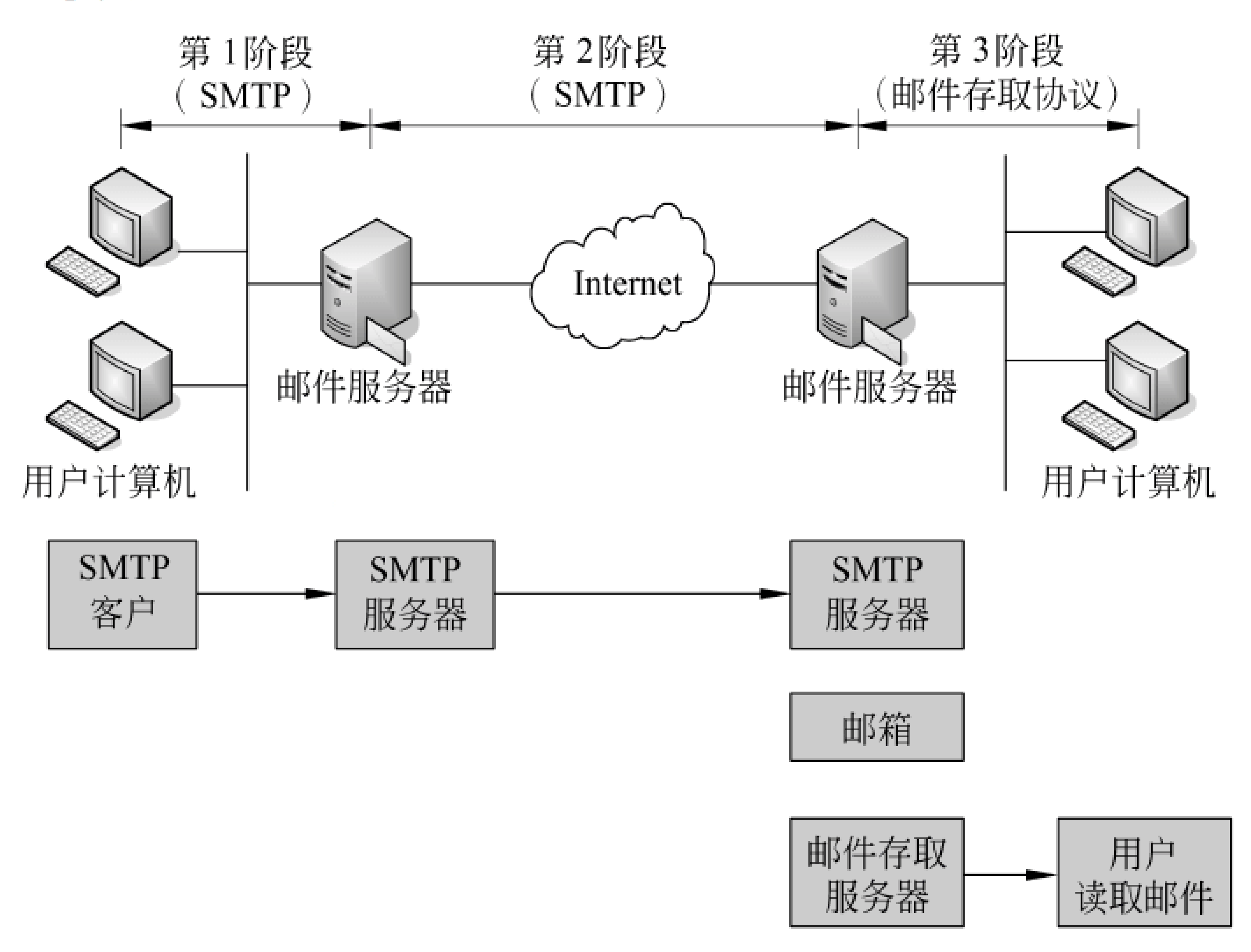


图 13-2　邮件传递过程示意图

(1) 在第 1 阶段,邮件报文从用户代理传送到本地服务器。用户使用的是 SMTP 客户程序,服务器使用的是 SMTP 服务器程序,邮件报文存放在本地服务器中。

(2) 在第 2 阶段,本地邮件服务器作为 SMTP 客户将邮件转发给作为 SMTP 服务器的远程服务器,直至到达目的地址所在的服务器。最终的邮件服务器将邮件报文存放到用户的个人邮箱中,等待用户读取。

(3) 在第 3 阶段,接收邮件的用户通过远程用户代理,使用 POP3 或 IMAP 对个人邮箱进行访问,读取邮件报文。

13.3.3 SMTP 协议

在 TCP/IP 协议族中,支持 Internet 电子邮件服务的基本协议是简单邮件传输协议(Simple Mail Transfer Protocol,SMTP)。SMTP 邮件传输协议为主机与邮件服务器(以及邮件服务器与邮件服务器)之间进行通信定义了各种通信命令及应答规则,其基本内容包括在 RFC821 与 RFC2821 中。

1. SMTP 命令和应答

SMTP 利用一些命令和应答在 MTA 客户和 MTA 服务器之间传输报文。图 13-3 给出了 SMTP 的命令和响应的关系示意图。表 13-1 和表 13-2 分别给出了 SMTP 的主要命令和应答的意义。



图 13-3 SMTP 的命令和响应示意图

表 13-1 SMTP 的主要命令

关 键 词	参数及说明	关 键 词	参数及说明
HELO	发送端的主机名	RSET	中止当前的邮件处理
MAIL FROM	发信人	VRFY	需要验证的收信人的名字
RCPT TO	预期的收件人	EXPN	需要扩展的邮件发送清单
DATA	邮件的主体	HELP	命令名
QUIT	结束会话	SEND FROM	预期的收信人

表 13-2 SMTP 的主要应答

代码	说 明	代码	说 明
220	服务就绪	450	邮箱不可使用
221	服务关闭传输通道	500	语法错,不能识别命令
250	请求命令完成	502	命令未实现
251	用户不是本地的,报文将被转发	552	所请求的动作异常终止,超过存储位置
354	开始邮件输入	553	所请求的动作未发生,邮箱名不允许使用

2. SMTP 协议的客户端与服务器端的交换过程

SMTP 协议的客户端与服务器端的交换过程可以分为 3 个阶段：连接建立、邮件传送和连接终止。

连接建立的大致过程如图 13-4 所示。

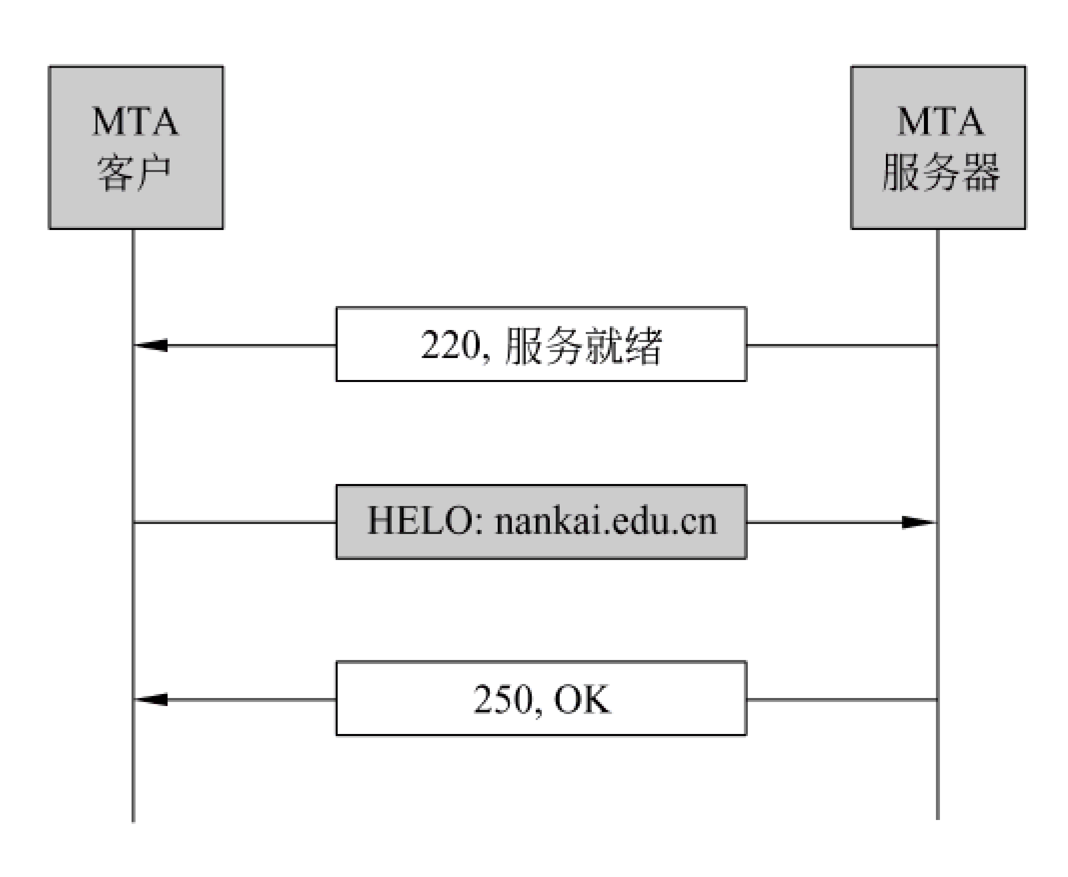


图 13-4 连接建立过程示意图

(1) 客户端使用 TCP 协议与服务器的 25 端口建立连接，SMTP 服务器进行响应。如果 SMTP 服务器接受客户端的连接请求，则将向 SMTP 客户端回送应答码“220”，表示该服务器可以提供 SMTP 服务。

(2) 客户端收到应答码后，通过发送“HELO”命令启动客户端与服务器之间的会话。“HELO”命令用来向服务器端提供客户的标识信息并请求 SMTP 服务器提供邮件服务。此时，服务器端将回送应答码“250”，表示客户端请求建立的邮件服务会话已经实现。

在 SMTP 客户与 SMTP 服务器之间的连接建立之后，客户端就可以向服务器发送邮件报文了。其过程如图 13-5 所示。

(1) 客户端使用 “MAIL FROM：Sender@nku.cn”向服务器报告发信人的邮箱与域名。

(2) 服务器端利用“250”(请求命令完成)进行响应。

(3) 客户端使用“RCPT TO”命令向服务器报告收信人的邮箱与域名。

(4) 服务器端发送“250”(请求命令完成)进行响应。

(5) 客户端利用“DATA”命令对报文的传送进行初始化。

(6) 服务器端回送“354”(开始邮件输入)进行响应。

(7) 客户端向服务器发送邮件报文内容。其中只有“.”的行表示报文内容结束。

(8) 服务器端使用“250”(请求命令完成)进行响应。

客户端在完成一次邮件报文的传输过程中始终起控制作用。报文发送完毕后会终止 SMTP 会话过程(如图 13-6 所示)。

13.3.4 邮件报文格式

Internet 电子邮件报文格式应遵循 RFC822 和 MIME 规范。其中，RFC822 是最基本的格式规范，而 MIME 则是对 RFC822 的扩充。

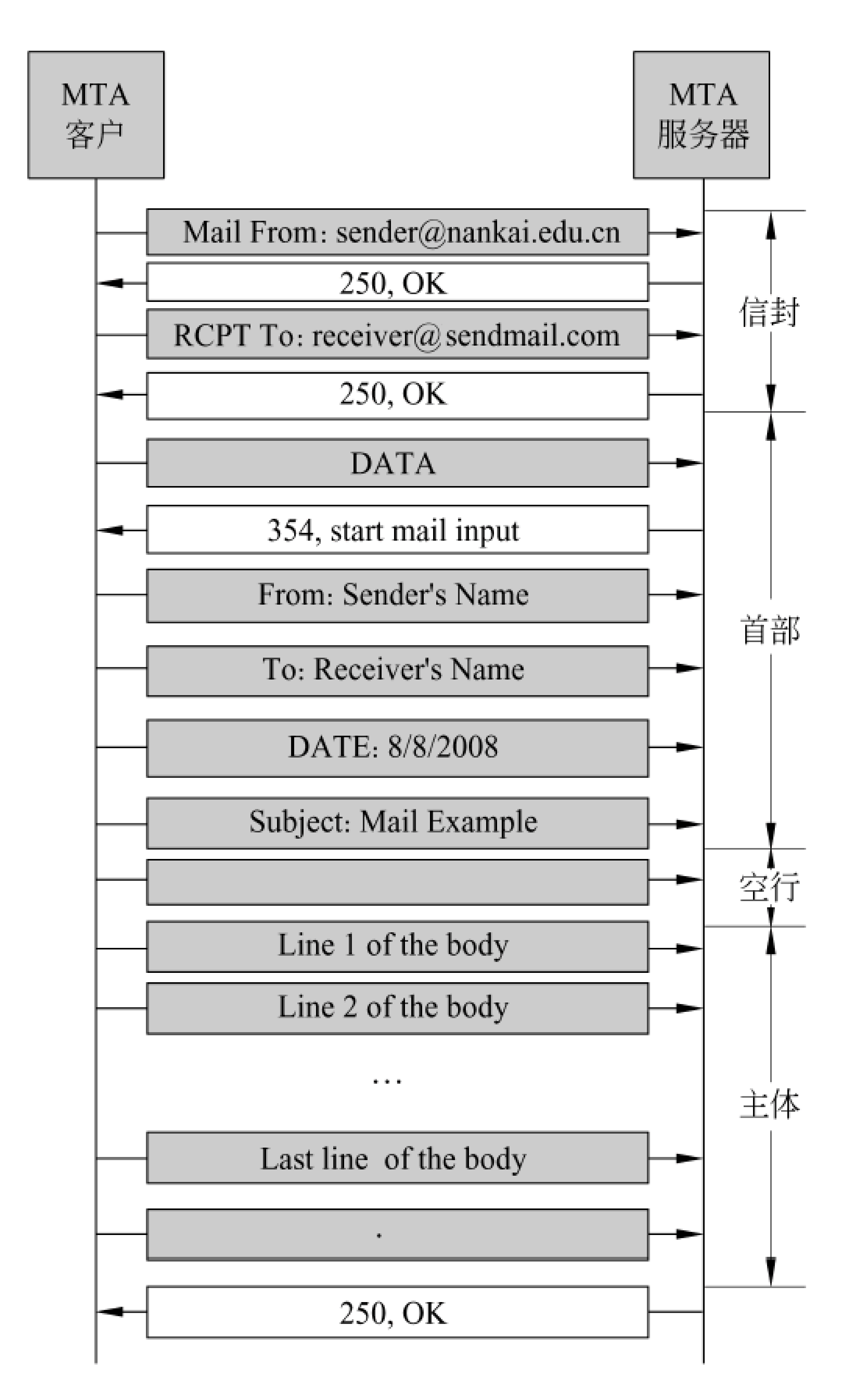


图 13-5 报文传送过程示意图

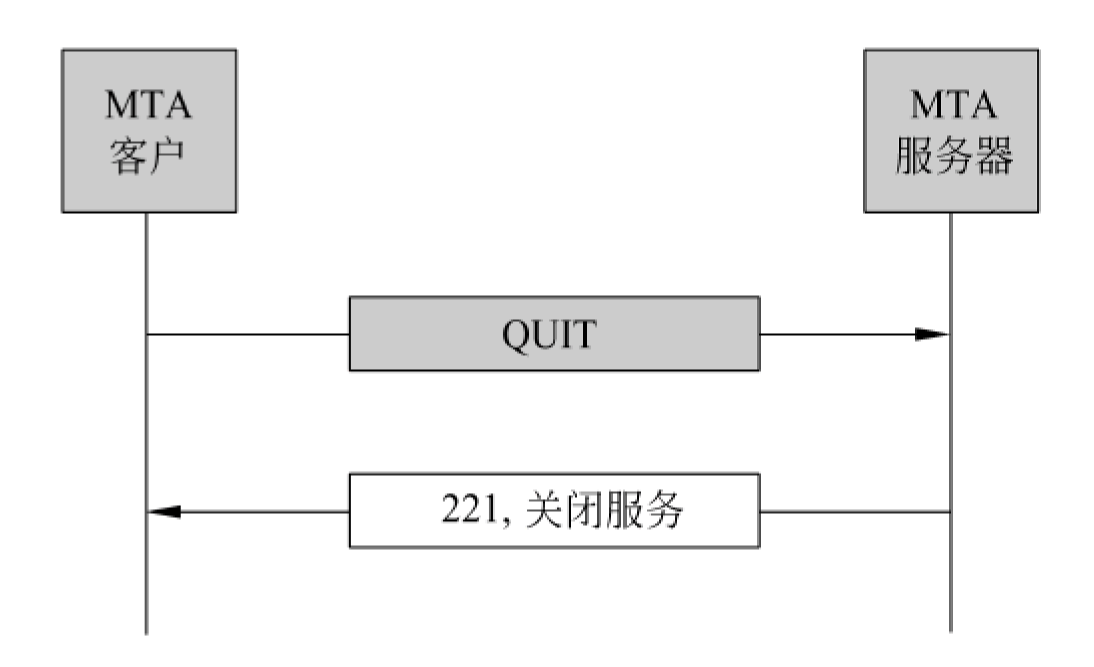


图 13-6 连接终止过程示意图

RFC822 格式规范的主要特点如下：

(1) 所有报文全部由 ASCII 码组成。

(2) 报文由报文行组成，各行之间用回车(CR)与换行(LF)符分隔。

(3) 报文中可包含多个报头字段和报头内容。

(4) 报文可包括跟随在报头后的报文体。如果报文中含有报文体，则报文体必须用一个空行与报头分隔。

由于 RFC822 只能发送包含 ASCII 字符的文本邮件，不支持语音、视频等多媒体邮件，因此具有一定的局限性。为了能使电子邮件系统传递多媒体等二进制邮件，1993 年提出了一种 Internet 邮件扩展协议 MIME (Multipurpose Internet Mail Extension)。MIME 是对 RFC822 的补充，本身并不能代替 RFC822。

在发送端，MIME 对非 ASCII 码格式的数据重新进行编码，其中编码方式支持 Base64、7bit、8bit、Quoted-printable、Binary 等标准。然后，将编码后的数据通过邮件传输代理 MTA 发送出去。在接收端，SMTP 服务器接收编码后的数据，然后由 MIME 还原成用户发送的原始数据。由此可见，MIME 是一个格式转换标准，符合 MIME 规范的软件能够将多媒体等非 ASCII 码格式的数据转换成 ASCII 码格式。

MIME 定义了 5 种邮件首部，用来扩充原有的 RFC822。这 5 种首部是 MIME-version (MIME 版本)、content-type (内容类型)、content-transfer-encoding (内容传输编码)、content-ID(内容标识)与 content-description(内容-描述)。与 MIME 协议相关的 RFC 文档包括 RFC2045～RFC2049 等 5 个，读者可根据需要自行查阅。

13.3.5　POP3 与 IMAP 协议

在邮件传递的 3 个阶段中，前两个阶段都采用了 SMTP 协议。但是，在第 3 个阶段，Internet 邮件系统并未采用 SMTP 协议。第 3 个阶段不采用 SMTP 的主要原因是 SMTP 协议采取“push”策略将邮件报文“推送”到服务器端。而在接收端，由于用户主机的开机状态不能保证，因此，“推送”策略并不是最好的。为此，Internet 邮件系统在第 3 阶段采取了“pull(拉)”的方式，用户在需要的时候主动从本地的邮件服务器下载自己的邮件。

从本地邮件服务器下载和读取邮件的协议主要包括第 3 版邮局协议 POP3(Post Office Protocol Version 3)与互联网消息访问协议 IMAP(Internet Message Access Protocol)。

1. POP3 协议

POP3 协议提供了一种将存储在本地 SMTP 服务器中的邮件传递到用户的机制，其相关的 RFC 文档包括 1939 和 RFC2449 等。

POP 的会话格式与 SMTP 类似，其通信过程大致可以描述为：当客户需要从邮件服务器中下载邮件时，客户端使用 TCP 协议与服务器的 110 端口建立连接。用户向服务器发送用户名和口令。在验证合法之后，用户端即可以列出邮件清单并能够逐个读取邮件。

POP 协议有两种工作模式：删除模式与保留模式。在读取邮件之后，删除模式把读过的邮件删除，而保留模式仍将读过的邮件保存在服务器中。

2. IMAP4 协议

IMAP 的功能比 POP3 更加强大，同时也更为复杂。与 IMAP 相关的 RFC 文档为 RFC2060。

与 POP3 协议相比，IMAP4 协议在以下 5 个方面进行了增强：

(1) 用户在下载邮件之前可以检查邮件的首部。

(2) 用户在下载邮件之前可以用特定的字符串搜索电子邮件的内容。

(3) 用户可以部分地下载电子邮件。
(4) 用户可以在邮件服务器上创建、删除邮箱或对邮箱更名。
(5) 为了存放电子邮件,用户可以在文件夹中创建分层次的邮箱。

13.3.6 Sendmail 简介

Sendmail 是 Linux/UNIX 环境下应用最广泛的邮件服务程序之一,它遵循 SMTP 协议,能够完成邮件的转发等功能。与此同时,Sendmail 还提供了一套“内容管理 API”,即 milter。milter 允许第 3 方程序在 MTA 处理邮件时,对邮件的内容进行检查甚至修改,而 Sendmail 则会根据第 3 方程序的返回值对 MTA 客户做出响应。因此,milter 在垃圾邮件过滤、病毒检测以及内容控制等方面都起着重要作用。

milter 提供的接口可以分为 5 类,分别是控制函数、数据访问函数、信息修改函数、其他句柄函数和回调函数。

1. 控制函数

控制函数及其功能描述如表 13-3 所示。

表 13-3 milter 的控制函数

函　数	功 能 描 述	函　数	功 能 描 述
smfi_opensocket	打开 socket 接口	smfi_setbacklog	定义监听队列大小
smfi_register	注册一个过滤器	smfi_setdbg	定义 milter 库的调试级别
smfi_setconn	指定所用的 socket	smfi_stop	按顺序关闭 milter
smfi_settimeout	设置超时时间	smfi_main	libmilter 主控函数

2. 数据访问函数

数据访问函数及其功能描述如表 13-4 所示。

表 13-4 milter 的数据访问函数

函　数	功能描述	函　数	功能描述
smfi_getsymval	取得 Sendmail 的宏值	smfi_setreply	设置 SMTP 的错误信息返回码
smfi_getpriv	取得私有数据指针	smfi_setmlreply	设置对 SMTP 多连接的错误信息返回码
smfi_setpriv	设置私有数据指针		

3. 信息修改函数

每个过滤器必须设置相应的标记,并将其传递给 smfi_register 之后才能调用相应的信息修改函数。否则,MTA 将认为回调函数发生错误并中止与过滤器的连接。信息修改函数及其功能描述如表 13-5 所示。

表 13-5　milter 的信息修改函数

函　数	功 能 描 述	需要设置的 SMFIF_ * flag
smfi_addheader	为消息添加头信息	SMFIF_ADDHDRS
smfi_chgheader	修改或删除消息的头信息	SMFIF_CHGHDRS
smfi_insheader	插入消息头信息	SMFIF_ADDHDRS
smfi_addrcpt	增加一个收件人	SMFIF_ADDRCPT
smfi_delrcpt	删除一个收件人	SMFIF_DELRCPT
smfi_replacebody	替换消息内容	SMFIF_CHGBODY

4. 其他句柄函数

其他句柄函数及其功能描述如表 13-6 所示。

表 13-6　milter 的句柄函数

函　　数	功　能　描　述
smfi_progress	通知 MTA 过滤器仍在处理邮件
smfi_quarantine	隔离一封邮件，需要原因

5. 回调函数

milter 回调函数主要用于在 SMTP 的传输过程中实现回调，它使用 smfi_register 进行注册。在 SMTP 会话的各个阶段，我们都可以通过回调函数方便地实现对邮件的任意操作，如接受、拒绝、丢弃、暂时拒绝及修改邮件信息等。垃圾邮件处理程序可以利用 milter 的回调功能实现对邮件内容的识别和过滤。SMTP 与 Milter 的调用关系如图 13-7 所示。表 13-7 给出了 SMTP 事务与回调函数的一一对应关系，表 13-8 给出了回调函数的返回值及其意义描述。

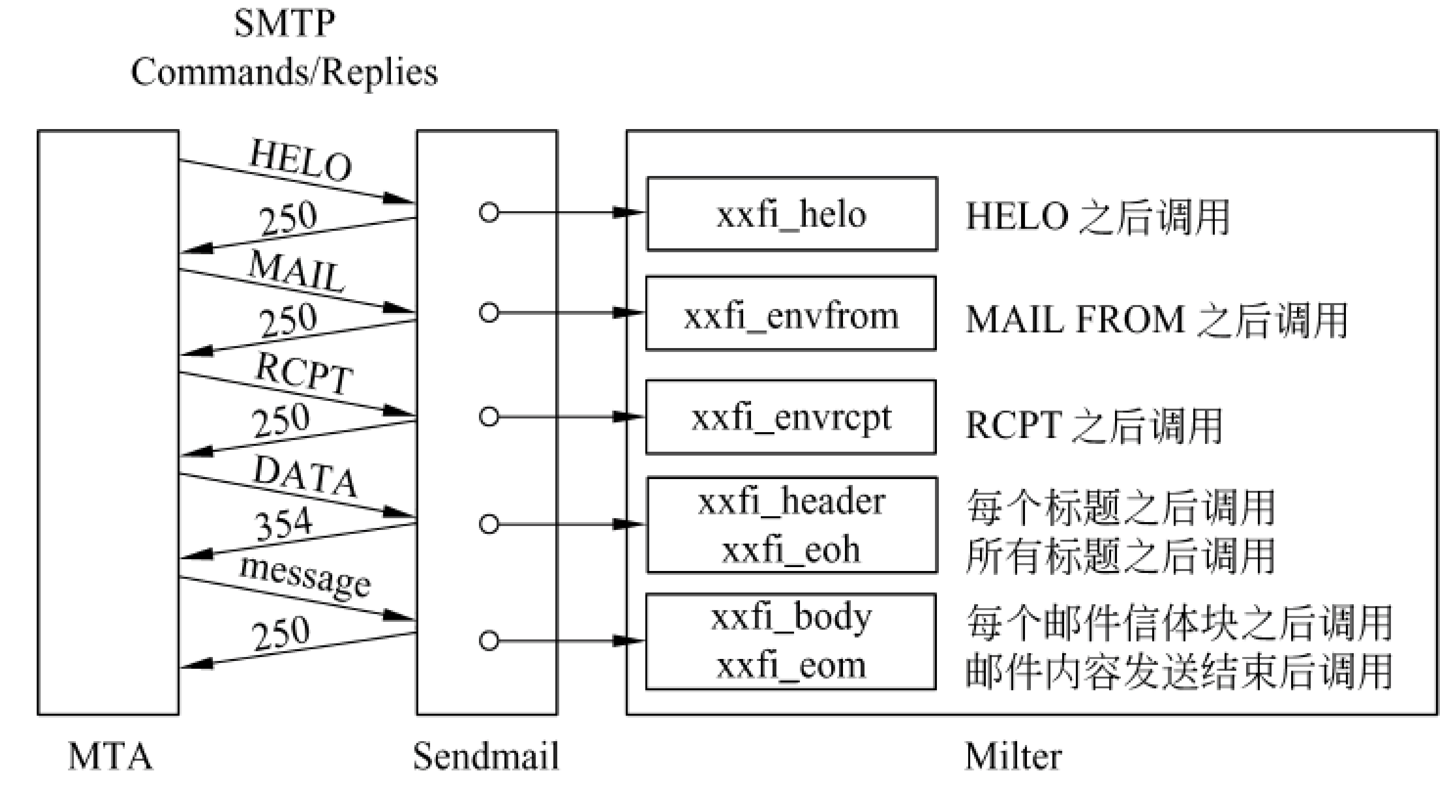


图 13-7　SMTP 各阶段对应的 milter 函数的示意图

表 13-7 SMTP 事务与回调函数的对应关系

SMTP 命令	milter 回调函数	SMTP 命令	milter 回调函数
(open SMTP connection)	xxfi_connect	DATA Header:...	xxfi_header
HELO...	xxfi_helo	[more headers]	[xxfi_header] xxfi_eoh
MAIL From:...	xxfi_envfrom	body...	xxfi_body
RCPT To:...	xxfi_envrcpt	[more body...]	[xxfi_body] xxfi_eom
[more RCPTs]	[xxfi_envrcpt]	QUIT (close SMTP connection)	xxfi_close

表 13-8 milter 回调函数的返回值

返回值	描　　述
SMFIS_CONTINUE	继续下一步操作
SMFIS_REJECT	对面向连接的程序接口,拒绝本次连接并调用 xxfi_close 对面向消息的程序接口(除 xxfi_eom 和 xxfi_abort 之外),拒绝这个邮件 对面向接收者的程序接口,拒绝当前接收者(但是继续传送本邮件)
SMFIS_DISCARD	对面向消息和接收者的程序接口,接收这个邮件,但悄悄地丢弃它(不通知发信人) 在面向连接的程序接口中不能把它作为返回值
SMFIS_ACCEPT	对面向连接的程序接口,接收本次连接,不再进行后面的过滤,并且调用 xxfi_close 对面向消息和接收者的程序接口,接收本邮件且不再进行后面的过滤
SMFIS_TEMPFAIL	返回一个临时的失败标记,比如 SMTP 将应答一个 4xx 的代码,表明临时的失败 对面向消息的程序接口(除了 xxfi_envfrom 之外),消息传送失败 对面向连接的程序接口,连接失败并调用 xxfi_close 对面向接收者的程序接口,仅对当前用户失败,邮件继续传送

13.4 编程训练设计分析

13.4.1 程序的流程

本节将利用 Sendmail 提供的 milter 回调函数对接收到的邮件进行黑/白名单过滤和正文关键字过滤。程序的主要流程如下(如图 13-8 所示):

(1) 开始接收邮件。

(2) 检查发信人是否在黑名单内,如果是,则拒绝处理该邮件;否则继续向下处理。

(3) 检查发信人是否在白名单内,如果是,则接受该邮件;否则继续向下处理。

(4) 如果发信人既不在黑名单内,又不在白名单内,则对邮件的内容进行关键字过滤。如果邮件中含有被过滤的关键词信息,则拒绝该邮件;否则接收该邮件。

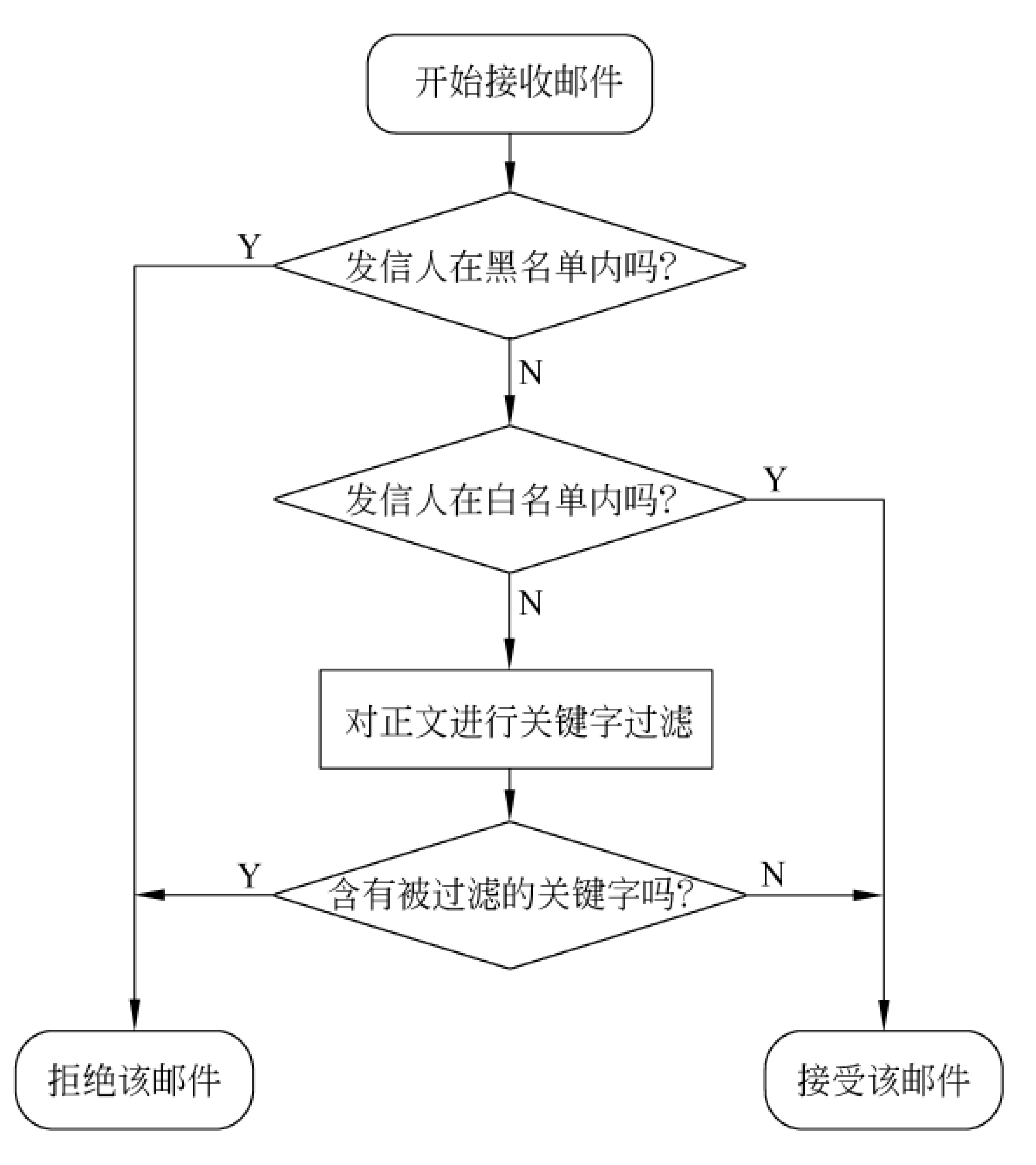


图 13-8　程序的流程示意图

13.4.2　程序的关键代码分析

1. 为每一个 SMTP 连接指定私有数据

在处理每一封邮件的过程中，Sendmail 都会创建一个 SMTP 连接，我们可以为需要使用的每个连接指定一个私有数据。私有数据类型的定义如下：

```
//为每一个连接定义一个私有数据类型，用来标识该连接所处理的邮件
typedef struct
{
    bool inWhiteList;            //如果邮件发送方在白名单内，则该值为 TRUE;否则为 FALSE
    bool inBlackList;            //如果邮件发送方在黑名单内，则该值为 TRUE;否则为 FALSE
} mlfipriv;
```

由上面关于 milter 回调函数的介绍可知，当建立 SMTP 连接时，milter 回调函数中的 xxfi_connect 将会被调用。因此，可以通过该函数将上面定义的私有数据指定给当前的 SMTP 连接。如果执行成功，则返回 SMFIS_CONTINUE，Sendmail 继续对邮件进行处理；否则返回 SMFIS_TEMPFAIL，Sendmail 向 MTA 客户发送一个临时的错误信号。主要程序代码如下所示：

```
//当客户 MTA 和服务端 MTA 建立 SMTP 连接时，该函数被调用
sfsistat
mlfi_connect(ctx,hostname,hostaddr)
    SMFICTX * ctx;
```

```
        char * hostname;
        _SOCK_ADDR * hostaddr;
    {
        //定义一个私有数据的指针,并为其分配内存空间
        struct mlfiPriv * priv;
        priv= (struct mlfipriv * )malloc(sizeof * priv);

        //如果分配内存失败,则返回一个临时的错误,表示现在不能进行处理
        if(priv==NULL)
        {
            return SMFIS_TEMPFAIL;
        }
        //如果分配内存成功,则对私有数据 priv进行初始化
        priv->inWhiteList=FALSE;
        priv->inBlackList=FALSE;

        //为当前连接保存私有数据
        if(smfi_setpriv(ctx,priv)==MI_SUCCESS)
        {
            //继续向下处理
            return SMFIS_CONTINUE;
        }
        else
        {
            //返回一个临时的错误
            return SMFIS_TEMPFAIL;
        }
    }
```

2. 黑名单过滤部分

当 MTA 客户向 Sendmail 发送“MAIL FROM”命令时,Sendmail 将会调用 milter 回调函数中的 xxfi_envfrom。由于“MAIL FROM”命令后面跟有发信人的地址,因此可以在函数 xxfi_envfrom 中实现黑/白名单的过滤功能。

如果发信人地址在黑名单内,则返回 SMFIS_REJECT,Sendmail 拒绝该邮件;否则继续向下处理。黑名单过滤部分的代码如下所示:

```
//获取当前连接的私有数据的指针
struct mlfiPriv * priv=MLFIPRIV;

//提取邮件发送方的地址
char * mail_addr=smfi_getsymval(ctx,"{mail_addr}");
FILE * fp_header=fopen("header","w");
fprintf(fp_header,"%s",mail_addr);
fclose(fp_header);
```

```
//读取黑名单,并判断当前的邮件发送方是否在黑名单内
FILE * fp_black= fopen("blacklist","r");

//如果成功打开文件,则将文件内被列入黑名单的地址与当前发送方的地址进行比较
if(fp_black!=NULL)
{
    //声明一个数组,来保存从黑名单文件中读取到的地址
    char black_addr[256];
    memset(black_addr,0,256);

    //从文件中每次读取一行,同发送方地址进行比较
    while(fgets(black_addr,256,fp_black)!=NULL)
    {
        if(strcmp(black_addr,mail_addr)==0)          //如果相等,则拒绝该邮件
        {
            //关闭文件
            fclose(fp_black);

            //拒绝该邮件
            return SMFIS_REJECT;
        }
    }

    //关闭文件
    fclose(fp_black);
}
```

代码开始的 MLFIPRIV 是定义的一个宏,用来获取当前连接所指定的私有数据的指针,其代码如下:

```
//为函数 smfi_getpriv(ctx)定义一个宏,该函数的作用是获取当前连接的私有数据的指针
#define MLFIPRIV     ((struct mlfiPriv * )smfi_getpriv(ctx))
```

3. 白名单过滤部分

如果发信人地址不在黑名单内,则检查其是否在白名单内。如果发信人地址在白名单内,则将私有数据部分的 inWhiteList 置为 TRUE;否则置为 FALSE。与黑名单过滤代码相同,白名单过滤部分的代码也需要在 milter 的回调函数 xxfi_envfrom 中实现。具体代码如下:

```
//如果发送方不在黑名单内,则继续读取白名单,并判断当前邮件的发送方是否在白名单内
FILE * fp_white= fopen("whitelist","r");

//如果成功打开文件,则将文件内被列入白名单的地址与当前发送方的地址进行比较
if(fp_white!=NULL)
```

```
{
    //声明一个数组,来保存从白名单文件中读取到的地址
    char white_addr[256];
    memset(white_addr,0,256);

    //从文件中每次读取一行,同发送方地址进行比较
    while(fgets(white_addr,256,fp_white)!=NULL)
    {
        if(strcmp(white_addr,mail_addr)==0)          //如果相等,则修改私有数据的相关内容
        {
            priv->inWhiteList=TRUE;
            break;
        }
    }

    //关闭文件
    fclose(fp_white);
}

//保存私有数据
if(smfi_setpriv(ctx,priv)==MI_SUCCESS)
{
    //继续进行处理
    return SMFIS_CONTINUE;
}
else
{
    //返回一个临时的错误
    return SMFIS_TEMPFAIL;
}
```

4. 关键字过滤部分

在回调函数 xxfi_body 中,可以获得邮件的正文。为了进行关键字过滤,首先需要对邮件正文的格式进行分析。为了简单起见,本实验仅要求针对 text/plain 格式的邮件正文进行处理。由于邮件正文通常经过 base64 或 quoted-printable 编码,因此,在进行关键字过滤之前需要进行解码处理。有关 base64 或 quoted-printable 解码算法,请读者自行查阅相关资料,这里不再给出相应的代码示例。

在获得解码后的正文之后,需要在字符串中查找是否含有相应的关键字。BM 算法是一种较为经典和常用的多关键字匹配算法,由 Bob Boyer 和 J Strother Moore 在 1977 年提出,其主要特点是匹配速度快。在下面给出的 BM 算法实现代码中,如果找到了相应的关键字,则返回它第一次出现的位置(偏移量);否则返回−1(代码如下所示)。

```
//功能:检查正文串中是否出现了样本串
```

```
//参数：text 为正文串；text_length 为正文串的长度；pattern 为样本串，pattern_length 为样本
//串的长度
//返回值：如果检索到样本串，则返回它在正文串中的第一次出现位置的下标；否则返回-1
int
bm_stringmatch(text,text_length,pattern,pattern_length)
    unsigned char * text;
    int text_length;
    unsigned char * pattern;
    int pattern_length;
{
    //定义变量 i、j、k。其中 i 表示当前样本串末端在正文串中的位置（即已经匹配的位置），j 表
    //示样本串中正在进行匹配的字符的下标；k 表示正文串中正在进行匹配的字符的下标
    int i,j,k;
    i=pattern_length-1;

    while(i<text_length)                //如果没有到正文的末尾，则继续匹配
    {
        j=pattern_length-1;
        k=i;

        //从右向左进行逐字符匹配
        while((j>=0)&&(pattern[j]==text[k]))
        {
            j--;
            k--;
        }

        if(j<0)
        {
            //如果匹配成功，则返回样本串在正文中的下标
            return i-pattern_length+1;
        }
        else
        {
            //如果本次匹配不成功，则将样本串右移，并进行下一次匹配
            i=i+bm_dist(pattern,pattern_length,text[k]);
        }
    }

    //如果没有找到样本串，则返回-1
    return-1;
}
```

其中，bm_dist 函数用来计算当一次匹配失败时样本需要右移的偏移量，其代码如下：

```
//功能：计算样本串应该右移的距离
```

```
//参数：pattern为样本串;pattern_length为样本串的长度;ch为匹配过程中第一个不相等的字符。
//返回值：样本串右移的距离
int bm_dist(pattern,pattern_length,ch)
    unsigned char * pattern;
    int pattern_length;
    unsigned char ch;
{
    int i;
    for (i=pattern_length-1;i>=0;i--)
    {
        if(ch==pattern[i])
        {
            return pattern_length-i-1;
        }
    }
    return pattern_length;
}
```

5. 主函数的代码分析

在主函数中首先需要定义一个结构体,用于指定 SMTP 各个阶段所对应的回调函数。该结构体将在注册过滤器时被使用。其代码如下所示:

```
struct smfiDesc smfilter=
{
    "milter",//filter name
    SMFI_VERSION,//version code--do not change
    SMFIF_CHGHDRS,
            //flags
    mlfi_connect,//connection info filter
    mlfi_helo,//SMTP HELO command filter
    mlfi_envfrom,//envelope sender filter
    mlfi_envrcpt,//envelope recipient filter
    mlfi_header,//header filter
    mlfi_eoh,//end of header
    mlfi_body,//body block filter
    mlfi_eom,//end of message
    mlfi_abort,//message aborted
    mlfi_close,//connection cleanup
    mlfi_unknown,//unknown SMTP commands
    mlfi_data,//DATA command
    mlfi_negotiate //Once,at the start of each SMTP connection
};
```

在 main 函数内部,需要对程序的参数进行检查。如果参数合法,则创建 SMTP 连接。具体代码如下:

```
bool setconn=FALSE;
int c;
const char * args="p:";
extern char * optarg;

//判断传入的参数是否合法,如果合法,则创建 SMTP 连接
while((c=getopt(argc,argv,args))!=-1)
{
    switch(c)
    {
      case'p':
        if(optarg==NULL|| * optarg=='\0')
        {
            (void)fprintf(stderr,"Illegal conn: %s\n",optarg);
            exit(EX_USAGE);
        }
        if(smfi_setconn(optarg)==MI_FAILURE)
        {
            (void)fprintf(stderr,"smfi_setconn failed\n");
            exit(EX_SOFTWARE);
        }
        setconn=TRUE;
        break;

      default:
          usage(argv[0]):
          exit(EX_USAGE);
    }
}

//如果创建连接失败,则输出提示信息,然后退出
if(!setconn)
{
    fprintf(stderr,"%s: Missing required -p argument\n",argv[0];
    usage(argv[0]);
    exit(EX_USAGE);
}
```

如果连接创建成功,则注册过滤器,然后调用 smfi_main 函数进入主控循环。代码如下:

```
//注册过滤器
if(smfi_register(smfilter)==MI_FAILURE)
{
    fprintf(stderr,"smfi_register failed\n");
    exit(EX_UNAVAILABLE);
}
```

```
//对 libmiter 的事件循环进行控制
return smfi_main();
```

13.5 扩展与提高

前边介绍了几种最基本的垃圾邮件过滤方法，比如黑名单过滤方法、白名单过滤方法和关键字过滤方法等。但是这几种方法都属于静态的垃圾邮件过滤技术，垃圾邮件的发送者只需要对邮件稍加变化就可能使这些过滤方法失效。同时，单纯依靠某几个关键字判断是否为垃圾邮件，难免会产生误判或者漏判。比如对于普通用户来讲，一封含有关键字“发票”的邮件很可能是一封垃圾邮件，但是在销售行业，这样的邮件却是很正常的。那么，有没有办法能根据邮件的整体状况进行更为智能的判断呢？有没有办法让过滤技术自动学习垃圾邮件与正常邮件的特点呢？有没有办法能针对不同的用户提供个性化的定制方案呢？为了解决这些问题，人们设计出了很多算法，其中应用最为广泛的就是贝叶斯算法。

13.5.1 贝叶斯算法

贝叶斯理论由英国数学家 Thomas Bayes 创立，它的基本思想是利用已经发生的事件预测未来事件发生的可能性。贝叶斯理论假设：如果事件的结果不确定，那么量化它的唯一方法就是事件的发生概率；如果过去试验中事件的出现概率已知，那么就可以用数学方法计算出未来试验中事件发生的概率。

将贝叶斯理论应用在垃圾邮件过滤技术中，可以得到这样的结论：如果某些关键字经常出现在垃圾邮件中，却很少出现在正常邮件中，那么当收到一封含有这些关键字的邮件时，它是垃圾邮件的可能性就会很大。

利用贝叶斯算法对垃圾邮件进行过滤包括两个主要过程，一是对垃圾邮件和正常邮件进行学习，二是根据学习的结果对收到的邮件进行分类和过滤。

对垃圾邮件和正常邮件进行学习的主要目的是为了了解各个关键字在这两种邮件中出现的概率。这一过程包括以下几个步骤。

(1) 收集大量的垃圾邮件(用户不想要的)和正常邮件(用户想要的)，并建立垃圾邮件集和正常邮件集。

(2) 提取邮件主题和邮件体中的独立字符串(如 ABC、IP 地址、域名等等)作为 TOKEN 串，并统计提取出的 TOKEN 串出现的次数(即字频)。使用该方法对垃圾邮件集以及正常邮件集中的所有邮件进行处理。

(3) 每一类邮件集对应一个哈希表，hashtable_good 对应正常邮件集，hashtable_bad 对应垃圾邮件集，在哈希表中存储 TOKEN 串到字频的映射关系。

(4) 计算每个哈希表中 TOKEN 串出现的概率，即 P=某 TOKEN 串的字频/对应哈希表的长度。

(5) 综合考虑 hashtable_good 和 hashtable_bad，并判断当新来的邮件出现某个 TOKEN 串时，该新邮件为垃圾邮件的概率。用 A 事件表示收到的新邮件为垃圾邮件，T1、T2、…、Tn 代表不同的 TOKEN 串，则 $P(\mathrm{A}|\mathrm{Ti})$就表示当新邮件中出现 TOKEN 串 Ti 时，

该邮件为垃圾邮件的概率。设 $P1(Ti)$为 TOKEN 串 Ti 在 hashtable_good 中对应的值，$P2(Ti)$为 TOKEN 串 Ti 在 hashtable_bad 中对应的值，则 $P(A|Ti)=P2(Ti)/[P1(Ti)+P2(Ti)]$。

(6) 建立新的哈希表 hashtable_propability，表中存储 TOKEN 串 Ti 到 $P(A|Ti)$的映射关系。

至此，对垃圾邮件集和正常邮件集的学习过程结束。

当收到一封新的邮件时，首先从邮件主题和邮件体中提取出独立的字符串作为 TOKEN 串，处理过程与上面介绍的第(2)步相同；然后在哈希表 hashtable_propability 中查找不同的 TOKEN 串所对应的键值。

假设由新邮件中共得到 n 个 TOKEN 串，分别记为 T1、T2、…、Tn，它们在 hashtable_propability 中对应的值为 $P1$、$P2$、…、Pn，用 $P(A|T1,T2,\cdots,Tn)$来表示当邮件中同时出现 TOKEN 串 T1、T2、…、Tn 时，该邮件是垃圾邮件的概率，则由复合概率公式可知：$P(A|T1,T2,\cdots,Tn)=(P1*P2*\cdots*Pn)/[P1*P2*\cdots*Pn+(1-P1)*(1-P2)*\cdots*(1-Pn)]$。当 $P(A|T1,T2,\cdots,Tn)$超过预定的阈值时，就可以判断该邮件是垃圾邮件。

13.5.2　贝叶斯算法的优点

2003 年 5 月，BBC 专题报道称贝叶斯过滤技术可以达到 99.7%的垃圾邮件识别率，同时误判率极低，是目前最为有效的反垃圾邮件技术，该算法具有以下几方面的优点。

(1) 贝叶斯算法对邮件的所有内容进行分析，而不仅仅针对其中的某个关键字，并且它能判别邮件是垃圾邮件还是正常邮件。例如：包含“发票”字样的邮件不一定就是垃圾邮件，如果采用关键字过滤技术，显然难以达到理想的效果。而使用贝叶斯算法，既考虑了这些词在垃圾邮件中出现的概率，同时又考虑了它在正常邮件中出现的概率，综合考虑这些因素才做出判断，因此说，贝叶斯算法具备一定的智能。

(2) 贝叶斯算法具备自适应功能。通过学习新的垃圾邮件和合法邮件的样本，贝叶斯算法将能对抗最新的垃圾邮件，并且对变体字有奇效。比如，垃圾邮件发送者开始使用“*发*票*”来代替“发票”，这样能够绕过一般的关键字检查，除非“*发*票*”也被加到新的关键字中。然而对于贝叶斯算法，当它发现邮件中含有“*发*票*”时，由于正常邮件中从来没有发现这个词，因此它是垃圾邮件的可能性将急剧增加，“*发*票*”这个新词无疑成了垃圾邮件的指示器。

(3) 贝叶斯算法更加个性化。它能学习并理解用户对邮件的偏好。如前所述，“发票”一词对很多人来说都意味着是垃圾邮件，但是对销售行业则意味着是正常的邮件。贝叶斯算法能根据用户的这种偏好进行处理。

(4) 贝叶斯算法支持多语种或者说与编码无关。对于贝叶斯算法而言，它分析的是字串，无论它是字、词、符号还是别的什么，当然更与语言无关。

(5) 贝叶斯过滤器很难被欺骗。垃圾邮件发送者一般通过减少垃圾词汇或者在邮件中多掺杂一些好的词汇，企图绕过一般的邮件内容检查。但由于贝叶斯算法具有个性化色彩，要想成功地绕过贝叶斯的检查，就不得不对每个收件人的偏好进行研究，而这几乎是不可能实现的。

第14章 基于特征码的恶意代码检测系统的设计与实现

14.1 编程训练目的与要求

基于特征代码的恶意代码检测系统是一种对文件进行扫描检测判定是否含有恶意代码的安全软件。本章在系统分析恶意代码检测系统基本工作原理的基础上，以基于特征检测的恶意代码检测系统为对象，研究恶意代码检测系统的设计与软件编程方法。

本章训练的主要目的是：

(1) 掌握恶意代码的分类、主要文件格式和相关检测技术等基本概念和背景知识。

(2) 掌握基于特征的恶意代码检测系统的基本工作原理、设计与实现方法。

(3) 掌握使用 p3scan 和 Clam AntiVirus 组建邮件病毒拦截网关的软件编程方法。

本章训练要求如下：

(1) 学习 Clam AntiVirus 引擎的原理。

(2) 对 Clam AntiVirus 引擎进行扩展编程，实现简单的基于特征码匹配的恶意代码检测程序。

14.2 相关背景知识

14.2.1 恶意代码的定义与分类

1. 恶意代码的定义

1989 年，Howard L. Johnson 提出了恶意代码(Malicious Code)的概念，他认为恶意代码机制包括改变保护、强审计、代码限制以及异常的用户和系统操作等。恶意代码又称为恶意软件，英文表述可以是 Malicious Software (简称为 Malware)，或者是 Malicious Code (简称为 Malcode)。本文统一使用 Malware 表示恶意代码。

20 世纪 90 年代末，恶意代码的定义随着计算机与网络技术的发展而逐渐丰富。Grimes 将恶意代码定义为：经过存储介质和网络进行传播，从一台计算机系统到另外一台计算机系统，未经授权认证破坏计算机系统完整性的程序或代码。其中，非授权性和破坏性是恶意代码的两个主要特点。J. Bergeron 等人将恶意代码定义为：可以影响系统保密性、完整性、数据和控制流以及系统功能的一组代码。Christodorescu 与 Jha 描述恶意代码是

具有恶意目的的程序。McGraw 与 Morrisett 定义恶意代码是通过对软件系统的增加、修改或者删除操作，而有意破坏或影响系统功能的程序代码。微软公司将恶意代码定义为：当系统运行时，达到攻击者故意破坏目的的软件。例如计算机病毒、蠕虫和木马等。目前研究人员已经将“凡是人为编制的，干扰计算机正常运行并造成计算机软硬件故障，甚至破坏数据的计算机程序或指令集合”都认为是恶意代码。本文根据对恶意代码的新特点和发展趋势的研究，给出如下恶意代码的定义：恶意代码是指能够影响计算机操作系统、应用程序和数据的完整性、可用性、可控性和保密性的计算机程序或代码。主要包括计算机病毒、蠕虫和木马程序等。但是，目前从恶意代码的发展趋势来看，计算机病毒、蠕虫和木马程序在技术上正逐步交叉、融合。

2. 恶意代码的分类

恶意代码有多种分类方法，主要的分类方法有两种：按照恶意代码的加载机制进行分类和按照恶意代码的传播特点进行分类。

按照恶意代码的加载机制可以将其分为两类：自动加载与人工加载。前者可以随同正常的程序启动执行，或者利用系统、程序等漏洞而获得加载运行；后者需要人工调用才能加载执行。

按照恶意代码的传播特点可以将其分成可自我复制和不可自我复制两类。前者在程序运行后具有连续自我复制能力，可将自身复制给其他宿主；后者一般不具有连续复制能力，恶意代码加载运行后，一般只完成自身的安装。

病毒、蠕虫和木马的定义

(1) 计算机病毒

1994 年 2 月 18 日，我国正式颁布实施了《中华人民共和国计算机信息系统安全保护条例》，《条例》第 28 条明确指出：“计算机病毒，是指编制或者插入的破坏计算机功能或者毁坏数据，影响计算机使用，并能自我复制的一组计算机指令或者程序代码。”此定义在我国具有法律性及权威性。本文沿用此定义。其中，自我复制传播(传染)的功能是计算机病毒的主要特征，计算机病毒的宿主一般是各种可执行的文件。

(2) 蠕虫

蠕虫是可以通过网络等途径将自身的全部代码或部分代码通过网络复制、传播给其他网络节点的程序。它不同于计算机病毒，不需要文件宿主。由于蠕虫是通过网络进行大量复制传播的，因此可造成网络阻塞，甚至瘫痪。其中比较典型的蠕虫有 Code Red、Blaster 和邮件蠕虫 Sobig、Mydoom 等。

(3) 木马

特洛伊木马(简称木马)一词来源于古希腊神话中希腊人使用木马战略攻陷特洛伊城的故事。借用到网络安全技术中，通常是指通过伪装欺骗手段诱使用户激活自身，但不具有复制、传播能力的恶意程序。木马可造成计算机系统被远程控制，下载安装其他恶意代码，造成敏感信息被盗、删除或破坏系统等后果。

14.2.2　可执行文件结构介绍

1. ELF 文件

可执行连接格式(Executable and Linking Format,ELF)文件简称为 ELF,扩展名为 elf。可执行连接格式 ELF 是 UNIX 系统实验室(USL)作为应用程序二进制接口(Application Binary Interface,ABI)而开发和发布的。工具接口标准委员会(TIS)选择了正在发展中的 ELF 标准作为工作在 32 位 Intel 体系上不同操作系统之间可移植的二进制文件格式。可以从两个方面(即两个视图)对 ELF 文件结构进行分析,一个是装载运行角度,另一个是连接角度。ELF 文件格式结构如图 14-1 所示。

LinkingView	ExeutionView
ELF header	ELF header
Program header table *optional*	Program header table
Section 1	Segment 1
…	
Section 2	Segment 2
…	
…	…
Section header table	Section header table *optional*

图 14-1　ELF 文件格式结构示意图

ELF 文件头给出解读整个 ELF 文件的路径图,它是一个固定的结构。文件头的结构在系统头文件 elf.h 中定义如下:

```
#define EI_NIDENT (16)
typedef uint16_t Elf32_Half;
typedef uint32_t Elf32_Word;
typedef uint32_t Elf32_Addr;
typedef uint32_t Elf32_Off;
typedef struct {
unsigned char e_ident[EI_NIDENT];          //上文所说的 e_ident
Elf32_Half e_type;                         //文件类型
Elf32_Half e_machine;                      //机器类型
Elf32_Word e_version;                      //文件版本
Elf32_Addr e_entry;                        //程序入口虚地址
Elf32_Off e_phoff;                         //程序头表文件偏移
Elf32_Off e_shoff;                         //节头表文件偏移
Elf32_Word e_flags;                        //处理器相关的标志
Elf32_Half e_ehsize;                       //ELF 文件头大小
Elf32_Half e_phentsize;                    //程序头表每个表项的大小
Elf32_Half e_phnum;                        //程序头表的表项数目
Elf32_Half e_shentsize;                    //节头表每个表项的大小
Elf32_Half e_shnum;                        //节头表的表项数目
Elf32_Half e_shstrndx;                     //节名字符串的节头表表项索引
} Elf32_Ehdr;
```

如果 e_type=1,表明它是重定位文件,可以从连接视图去解读它;如果 e_type=2,表明它是可执行文件,可以从装载运行视图中去解读它;如果 e_type=3,表明它是共享动态库文件,同样可以从装载运行视图中去解读它;如果 e_type=4,表明它是 Core dump 文件,则从哪个视图中去解读依赖于具体的实现。

每个程序头表(Program Header Table)的每个表项的结构定义如下:

```
typedef struct
{
Elf32_Word p_type;                    //段类型
Elf32_Off p_offset;                   //在文件中的偏移
Elf32_Addr p_vaddr;                   //执行时的虚地址
Elf32_Addr p_paddr;                   //执行时的物理地址
Elf32_Word p_filesz;                  //在文件中的字节数
Elf32_Word p_memsz;                   //在内存中的字节数
Elf32_Word p_flags;                   //标志
Elf32_Word p_align;                   //字节对齐
} Elf32_Phdr;
```

而节头表(section header table)的各个表项给出了如何从连接视图解读 ELF 文件的路径图。节头表每个表项的结构如下所示：

```
typedef struct
{
Elf32_Word sh_name;              //节名索引
Elf32_Word sh_type;              //节类型
Elf32_Word sh_flags;             //加载和读写标志
Elf32_Addr sh_addr;              //执行时的虚地址
Elf32_Off sh_offset;             //在文件中的偏移
Elf32_Word sh_size;              //字节大小
Elf32_Word sh_link;              //与其他节的关联
Elf32_Word sh_info;              //其他信息
Elf32_Word sh_addralign;         //字节对齐
Elf32_Word sh_entsize;           //如果由表项组成,每个表项的大小
} Elf32_Shdr;
```

2. PE 文件

可移植的可执行 PE(Portable Executable)文件是 Win32 环境自带的可执行文件格式。PE 文件的一些特性继承了 UNIX 的 Coff(common object file format)文件格式。PE 文件格式是跨 win32 平台的,即 Windows 运行在非 Intel 的 CPU 上,任何 win32 平台的 PE 装载器都能识别和使用该文件格式。移植到不同的 CPU 上 PE 执行文件会有一些改变。所有 win32 可执行文件（除了 VxD 和 16 位的 Dll)都使用 PE 文件格式,包括 NT 的内核模式驱动程序(kernel mode drivers)。PE 文件格式如图 14-2 所示。

PE 文件结构
MS-DOS MZ 头部
MS-DOS 实模式残余程序
PE 文件标志
PE 文件头
PE 文件 可选头部
.text段头部
.bss段头部
.rdata段头部
⋮
.debug段头部
.text段
.bss段
.rdata段
⋮
.debug段

图 14-2　PE 文件格式

PE 文件的装载过程大致为：

(1) 当 PE 文件被执行时,PE 装载器检查 DOS MZ header 里的 PE header 偏移量。如果找到,则跳转到 PE header。

(2) PE 装载器检查 PE header 的有效性。如有效,跳转到 PE header 的尾部。

(3) 紧跟 PE header 的是节表(section table)。PE 装载器读取其中的节信息,并采用文件映射方法将这些节映射到内存,同时附上节表里指定的节属性。

(4) PE 文件映射入内存后,PE 装载器将处理 PE 文件中类似 import table(引入表)的逻辑部分。

下面对于 PE 格式中几个重要的结构需做以下几点说明。

(1) MS-DOS 头部

PE 文件的第一个结构是 MS-DOS 头,该结构为了兼容旧的 DOS 程序设计的,如果一个 Win32 程序在 DOS 模式下运行(所谓的 DOS 模式是指纯 DOS 环境,而不是 Windows 控制台),DOS 头部会把执行定位到 MS-DOS 实模式残余程序,该程序会调用 int 21 中断输出一个字符串“This program cannot be run in DOS mode”,然后直接退出。

MS-DOS 头部在“winnt.h”里面有定义:

```
typedef struct _IMAGE_DOS_HEADER {          //DOS .EXE header
  WORD  e_magic;                            // Magic number
  WORD  e_cblp;                             //Bytes on last page of file
  WORD  e_cp;                               //Pages in file
  WORD  e_crlc;                             // Relocations
  WORD  e_cparhdr;                          // Size of header in paragraphs
  WORD  e_minalloc;                         // Minimum extra paragraphs needed
  WORD  e_maxalloc;                         // Maximum extra paragraphs needed
  WORD  e_ss;                               // Initial (relative) SS value
  WORD  e_sp;                               // Initial SP value
  WORD  e_csum;                             // Checksum
  WORD  e_ip;                               // Initial IP value
  WORD  e_cs;                               // Initial (relative) CS value
  WORD  e_lfarlc;                           // File address of relocation table
  WORD  e_ovno;                             // Overlay number
  WORD  e_res[4];                           // Reserved words
  WORD  e_oemid;                            // OEM identifier (for e_oeminfo)
  WORD  e_oeminfo;                          // OEM information; e_oemid specific
  WORD  e_res2[10];                         // Reserved words
  LONG  e_lfanew;                           // File address of new exe header
} IMAGE_DOS_HEADER, * PIMAGE_DOS_HEADER;
```

第一成员变量 e_magic,被称为魔术数字,它被用于表示一个 MS-DOS 兼容的文件类型。所有 MS-DOS 兼容的可执行文件都将这个值设为 0x5A4D,表示 ASCII 字符 MZ。MS-DOS 头部之所以有的时候被称为 MZ 头部,就是这个缘故。

至于其余的成员变量,基本上都是为了 DOS 下实模式设计,如今已经没有什么实际作用,除了最后一个成员变量:e_lfanew。这个成员变量用来表示 PE 头部在这个 PE 文件中的偏移量。

通过如下代码可以获得 PE 头部地址:

```
BYTE * pFileImage= (BYTE* )pPeImage;
```

```
PIMAGE_DOS_HEADER pDosHeader= (PIMAGE_DOS_HEADER)pFileImage;
PIMAGE_FILE_HEADER pFileHeader= (PIMAGE_FILE_HEADER) (pFileImage+ pDosHeader->e_lfanew+ 4);
```

注意在计算偏移地址的时候出了偏移量 pDosHeader->e_lfanew，还有一个 DWORD 的偏移，这个 DWORD 是存储 PE 文件标志的，值为 0x4550，对应 ASCII 字符“PE”。

(2) PE 头部

接下来的结构体是 PE 头部。PE 头部在“winnt.h”中定义如下：

```
typedef struct _IMAGE_FILE_HEADER {
  WORD   Machine;
  WORD   NumberOfSections;
  DWORD  TimeDateStamp;
  DWORD  PointerToSymbolTable;
  DWORD  NumberOfSymbols;
  WORD   SizeOfOptionalHeader;
  WORD   Characteristics;
} IMAGE_FILE_HEADER, * PIMAGE_FILE_HEADER;
```

这个结构比较简单，Machine 表示这个可执行文件被构建的目标机器种类。

NumberOfSection 表示本 PE 文件段数量。每一个段头部和段实体都在文件中连续地排列着，在定位段时，首先根据段数目确定全部段头部信息，然后根据段头部内部的信息依次定位段实体。

TimeDataStamp 是一个时间戳变量；PointerToSymbolTable 和 NumberOfSymbols 确定了符号表的位置和大小。

SizeOfOptionalHeader 表示选项头部的大小，选项头部就在 PE 文件头部后面线性排列，这个结构容后介绍，但是读者不要被名称迷惑，选项头部是对 PE 文件执行至关重要的结构，并非“Optional”。

Characteristics 表示了文件的一些特征。比如对于一个可执行文件而言，分离调试文件是如何操作的。

(3) 选项头

选项头在“winnt.h”中定义如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  // Standard fields.
  WORD   Magic;
  BYTE   MajorLinkerVersion;
  BYTE   MinorLinkerVersion;
  DWORD  SizeOfCode;
  DWORD  SizeOfInitializedData;
  DWORD  SizeOfUninitializedData;
  DWORD  AddressOfEntryPoint;
  DWORD  BaseOfCode;
  DWORD  BaseOfData;
  // NT additional fields.
```

```
    DWORD  ImageBase;
    DWORD  SectionAlignment;
    DWORD  FileAlignment;
    WORD   MajorOperatingSystemVersion;
    WORD   MinorOperatingSystemVersion;
    WORD   MajorImageVersion;
    WORD   MinorImageVersion;
    WORD   MajorSubsystemVersion;
    WORD   MinorSubsystemVersion;
    DWORD  Win32VersionValue;
    DWORD  SizeOfImage;
    DWORD  SizeOfHeaders;
    DWORD  CheckSum;
    WORD   Subsystem;
    WORD   DllCharacteristics;
    DWORD  SizeOfStackReserve;
    DWORD  SizeOfStackCommit;
    DWORD  SizeOfHeapReserve;
    DWORD  SizeOfHeapCommit;
    DWORD  LoaderFlags;
    DWORD  NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, * PIMAGE_OPTIONAL_HEADER32;
```

此结构成员按照功能的区别可以分为两个域：标准域和 NT 附加域。标准域由前 9 个成员变量构成，是和 UNIX 可执行文件的 COFF 格式所公共的部分。虽然标准域保留了 COFF 中定义的名字，但是 Windows NT 仍然将它们用做不同的目的。

其中，Magic 标示了不同 PE 文件的种类，一般的 Win32 程序都是 0x10b；MajorLinkerVersion、MinorLinkerVersion：表示链接此映像的链接器版本；SizeOfCode：可执行代码尺寸；SizeOfInitializedData：已初始化的数据尺寸；SizeOfUninitializedData：未初始化的数据尺寸；AddressOfEntryPoint：本 PE 文件执行的入口点；BaseOfCode：已载入映像的代码（“. text”段）的相对偏移量；BaseOfData：已载入映像的未初始化数据（“. bss”段）的相对偏移量。其他变量具体含义请参见表 14-1 所示。

其中，RVA 代表相对虚拟地址，类似文件的偏移量。

（4）PE 文件段

PE 文件的段没有什么特定的结构特点，它几乎可以被链接器链接到 PE 文件的任何地方，程序执行时从 PE 文件定位段全靠段头部。

段头部每个 40 字节长，以数组的形式存放在 Image Optional Header 后面，可以使用如下代码获得该数组的起始地址：

```
PIMAGE_SECTION_HEADER pSectionHeader= (PIMAGE_SECTION_HEADER) (((char* )
pOptionalHeader)+ sizeof (IMAGE_OPTIONAL_HEADER32));
```

表 14-1　选项头结构

字　段	含　义
AddressOfEntryPoint	PE 装载器准备运行的 PE 文件的第一个指令的 RVA。若要改变整个执行的流程，可以将该值指定到新的 RVA，这样新的 RVA 处的指令首先被执行
ImageBase	PE 文件的优先装载地址。比如，如果该值是 400000h，PE 装载器将尝试把文件装到虚拟地址空间的 400000h 处。"优先"表示若该地址区域已被其他模块占用，PE 装载器会选用其他空闲地址
SectionAlignment	内存中节对齐的粒度。例如，如果该值是 4096 (1000h)，那么每节的起始地址必须是 4096 的倍数。若第一节从 401000h 开始且大小是 10 个字节，则下一节必定从 402000h 开始，即使 401000h 和 402000h 之间还有很多空间没被使用
FileAlignment	文件中节对齐的粒度。例如，如果该值是(200h)，则每节的起始地址必须是 512 的倍数。若第一节从文件偏移量 200h 开始且大小是 10 个字节，则下一节必定位于偏移量 400h，即使偏移量 512 和 1024 之间还有很多空间没被使用/定义
MajorSubsystem Version MinorSubsystem Version	win32 子系统版本。若 PE 文件是专门为 Win32 设计的，则该子系统版本必定是 4.0，否则对话框不会有 3 维立体感
SizeOfImage	内存中整个 PE 映像体的尺寸。它是所有头和节经过节对齐处理后的大小
SizeOfHeaders	所有头＋节表的大小，即等于文件尺寸减去文件中所有节的尺寸。可以以此值作为 PE 文件第一节的文件偏移量
Subsystem	NT 用来识别 PE 文件属于哪个子系统。对于大多数 Win32 程序而言，只有两类值：Windows GUI 和 Windows CUI(控制台)
DataDirectory	一 IMAGE_DATA_DIRECTORY 结构数组。每个结构给出一个重要数据结构的 RVA，比如引入地址表等

可以读取 PE 文件头部的 NumberOfSections 变量获取该数组的大小。

IMAGE_SECTION_HEADER 在"winnt.h"中的定义如下：

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE   Name[IMAGE_SIZEOF_SHORT_NAME];
         union {
         DWORDPhysicalAddress;
         DWORDVirtualSize;
  } Misc;
  DWORD  VirtualAddress;
  DWORD  SizeOfRawData;
  DWORD  PointerToRawData;
  DWORD  PointerToRelocations;
  DWORD  PointerToLinenumbers;
  WORD   NumberOfRelocations;
  WORD   NumberOfLinenumbers;
  DWORD  Characteristics;
} IMAGE_SECTION_HEADER, * PIMAGE_SECTION_HEADER;
```

- 其中，Name 存储的区段的名称，这个名称最大长度为 8 字节，且开头第一个字符必须是"."，如".text"、".data"等。
- Misc 为保留字段，用于兼容前期版本。

- VirtualAddress：这个域标识了进程地址空间中要装载这个段的虚拟地址。实际的地址由将这个域的值加上可选头部结构中的 ImageBase 虚拟地址得到。切记，如果这个映像文件是一个 DLL，那么这个 DLL 就不一定会装载到 ImageBase 要求的位置。所以一旦这个文件被装载进入了一个进程，实际的 ImageBase 值应该通过使用 GetModuleHandle 来检验。
- SizeOfRawData：表示原始数据的大小，也就是根据 FileAlignment 进行对齐之前的数据大小。
- PointerToRawData：这是一个文件中段实体位置的偏移量。
- 接下来的 4 个变量 PointerToRelocations、PointerToLinenumbers、NumberOfRelocations、NumberOfLinenumbers 在 PE 格式中不使用。
- Characteristics 定义了段的特征。

表 14-2 显示了不同 Characteristics 值对应的不同含义。

表 14-2 Characteristics 取值范围及含义

可能取值	对应含义	可能取值	对应含义
0x00000020	代码段	0x10000000	共享段
0x00000040	已初始化数据段	0x20000000	可执行段
0x00000080	未初始化数据段	0x40000000	可读段
0x04000000	该段数据不能被缓存	0x80000000	可写段
0x08000000	该段不能被分页		

14.2.3 恶意代码检测技术与发展趋势

虽然恶意代码判定十分困难，但是恶意代码的检测技术始终是网络安全研究的一个重点问题。恶意代码检测技术有多种分类方法，常用的方法有：按照部署方式分类、按照功能分类与按照检测数据类型分类。

1. 恶意代码检测部署方式

按照部署方式恶意代码检测技术又可以分为：基于主机的恶意代码检测方法和基于网络的恶意代码检测方法。

(1) 基于主机的恶意代码检测技术

基于主机的恶意代码检测技术包括：基于特征的扫描技术、校验和技术、沙箱技术与安全操作系统等。

- 最常用的恶意代码技术是基于特征的扫描。这种方法主要是源于模式匹配的思想。扫描程序工作之前，必须先建立恶意代码的特征文件，根据特征文件中的特征串，在扫描文件中进行匹配查找。用户通过更新特征文件扫描软件，查找最新的恶意代码版本。这种技术目前广泛地应用于反病毒引擎中。
- 校验和是一种保护信息资源完整性的控制技术，例如，Hash 值和循环冗余码等。只

要文件内部有一个比特改变,校验和值就会改变。未被恶意代码感染的系统首先会生成检测数据,然后周期性地使用校验和法检测文件的改变情况。运用校验和法检查恶意代码有 3 种基本的方法:

- 在恶意代码检测软件中设置校验和法:对检测的对象文件计算其正常状态的校验和,并将其写入被查文件或检测工具中,然后进行比较。
- 在应用程序中嵌入校验和法:将文件正常状态的校验和写入文件本身中,每当应用程序启动时,就比较现行校验和与原始校验和,实现应用程序的自我检测功能。
- 将校验和程序常驻内存:每当应用程序开始运行时,就自动比较检查应用程序内部或其他的文件中预留保存的校验和。

• 沙箱技术是指根据系统中每一个可执行程序的访问资源,以及系统赋予的权限,建立应用程序的"沙箱",限制恶意代码的运行。每个应用程序都运行在自己受保护的"沙箱"之中,不能影响其他程序的运行。同样,这些程序的运行也不能影响操作系统的正常运行,操作系统与驱动程序也被保存在各自的"沙箱"之中。美国加州大学 Berkeley 实验室开发了基于 Solaris 操作系统的沙箱系统,应用程序经过系统底层调用解释执行,系统会自动判断应用程序调用的底层函数是否符合系统的安全要求,以此决定是否执行。

  对于每个应用程序,沙箱都为其准备了一个配置文件,限制该文件能够访问的资源与系统赋予的权限。Windows XP/2003 操作系统提供了一种软件限制策略,隔离具有潜在危害的代码。这种隔离技术其实也是一种沙箱技术,可以保护系统免受通过电子邮件和互联网传染的各种恶意代码的侵害。这些策略允许选择系统管理应用程序的方式:应用程序既可以被"限制运行",也可以被"禁止运行"。通过在"沙箱"中执行不受信任的代码与脚本,系统可以限制甚至防止恶意代码对系统完整性的破坏。

• 恶意代码成功入侵的重要一环是获得系统的控制权,使操作系统为它分配系统资源。无论类别与目的如何,恶意代码都必须具有相应的权限。没有足够的权限,恶意代码不可能实现其预定的恶意目标,或者仅能够实现其部分恶意目标。安全操作系统就是对所有要运行的代码进行安全审核,防止恶意代码获得系统的控制权,以达到防范恶意代码的目的。

(2) 基于网络的恶意代码检测技术

基于网络的恶意代码检测技术包括:基于 GrIDS 的恶意代码检测、基于 PLD 硬件的检测、基于 HoneyPot 的检测与基于 CCDC 的检测。

• GrIDS 主要是针对大规模网络攻击和自动化入侵设计的,它收集计算机和网络活动的数据以及它们之间的连接,在预先定义的模式库的驱动下,将这些数据构建成网络活动行为来表征网络活动结构上的因果关系。

  GrIDS 通过建立和分析节点间的行为图(activity graph),通过与预定义的行为模式图进行匹配,检测恶意代码是否存在,是当前检测分布式恶意代码入侵的有效工具之一。

• 华盛顿大学应用研究室的 John W. Lockwood、James Moscola1 和 MatthewKulig 等。提出了一种采用可编程逻辑设备(Programmable Logic Devices,PLDs)对抗恶

意代码的防范系统。该系统由3个相互内联部件DED(Data Enabling Device)、CMS(Content Matching Server)和RTP(Regional Transaction Processor)组成。其中DED负责捕获流经网络出入口的所有数据包,根据CMS提供的特征串或规则表达式对数据包进行扫描匹配,并把结果传递给RTP;CMS负责从后台的MYSQL数据库中读取已经存在的恶意代码特征,编译综合成DED设备可以利用的特征串或规则表达式;RTP根据匹配结果决定DED将采取何种操作。在恶意代码大规模入侵时,系统管理员首先把该恶意代码的特征添加到CMS的特征数据库中,DED扫描到相应特征才会请求RTP做出放行或阻断的响应。

- 早期HoneyPot检测方法主要用于防范网络黑客攻击。ReVirt是能够检测网络攻击或网络异常行为的HoneyPot系统。Spitzner首次运用HoneyPot防御恶意代码攻击。

  HoneyPot之间可以相互共享捕获的数据信息,采用NIDS的规则生成器产生恶意代码的匹配规则,当恶意代码根据一定的扫描策略扫描存在漏洞主机的地址空间时,HoneyPot可以捕获恶意代码扫描攻击的数据,然后采用特征匹配来判断是否有恶意代码攻击。
- 由于主动式传播恶意代码具有生物病毒特征,美国安全专家提议建立CCDC(The Cyber Centers for Disease Control)来对抗恶意代码攻击。

  防范恶意代码攻击的CCDC体系具备以下功能:
  - 鉴别恶意代码的爆发期。
  - 分析恶意代码的样本特征。
  - 对抗恶意代码的传染。
  - 预测恶意代码新的传染途径。
  - 开展恶意代码对抗工具的前瞻性研究以对抗未来恶意代码的威胁。

CCDC能够实现对大规模恶意代码入侵的预警、防御和阻断。但CCDC体系也存在一些问题。这些问题主要表现在以下几个方面:

① CCDC是一个规模庞大的防范体系,系统运转的代价高。

② 由于CCDC体系的开放性,CCDC自身的安全问题也不容忽视。

③ 在CCDC防范体系中,攻击者能够监测恶意代码攻击的全过程,通过深入理解CCDC体系防范恶意代码的工作机制,可能研究出突破CCDC防范体系的恶意代码。

2. 恶意代码检测技术功能分类

卡巴斯基实验室Alisa Shevchenko认为恶意代码检测技术由两部分组成——技术组件和分析组件,其结构如图14-3所示。

(1) 技术组件

病毒软件(或其他安全软件)会采取相应的行动:通知用户,询问处理方式,将文件隔离,阻断未认证的程序行为等。

恶意代码的特征主要表现在以下三个方面:一是它是具有一定内容的文件,二是在操作系统中进行的一组行为,三是操作系统中作用效果的集合。因此对恶意代码的识别可以从不同的层面上进行:通过字节序列、通过行为和通过对操作系统的影响等。一般可以采

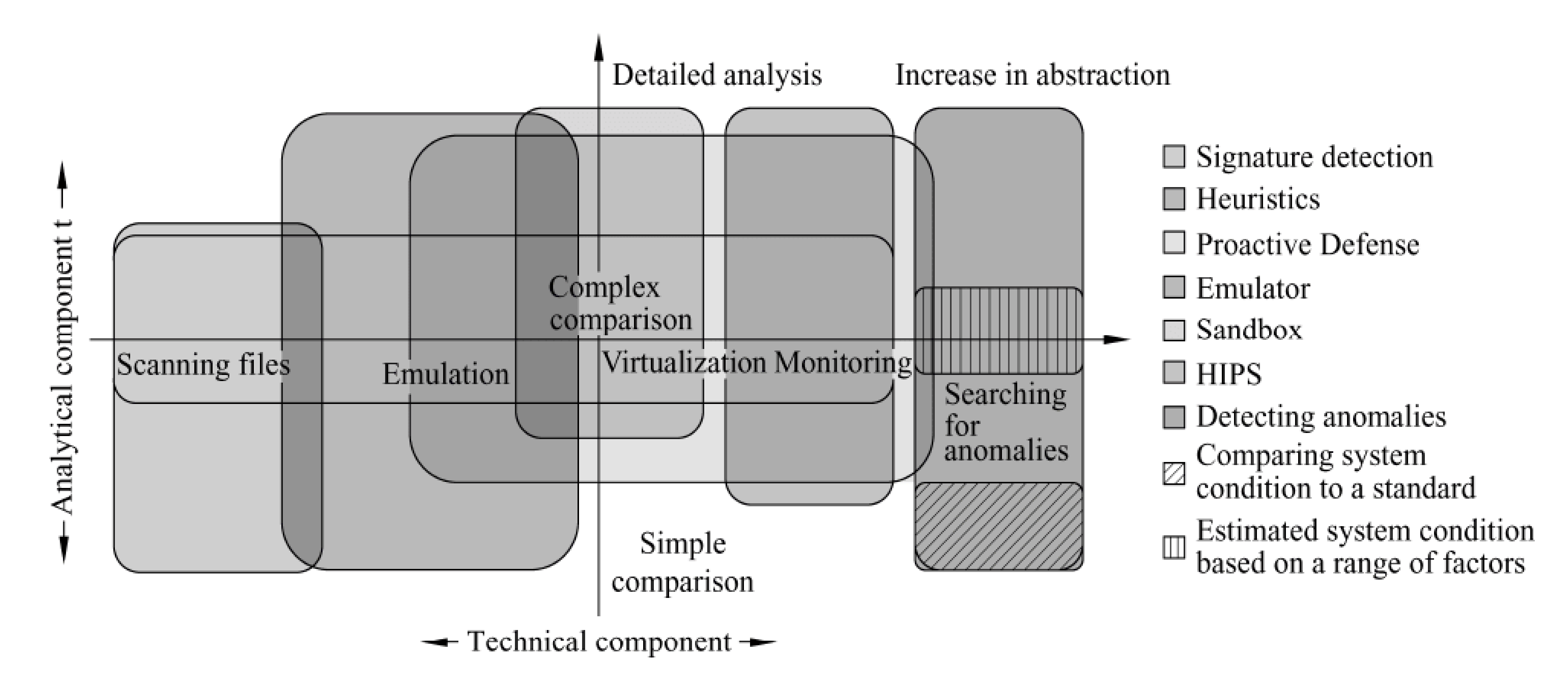


图 14-3 恶意代码检测技术的技术组件与分析组件结构示意图

用以下检测技术进行：将文件作为字节序列处理；模拟程序代码、在沙箱中运行程序(sandbox)或其他类似的虚拟技术、监视系统事件与搜索系统异常。

上述方法是按处理代码时抽象级别由低到高排列的。这里抽象级别的意思是研究从何种角度来查看可执行程序：作为原始的数字对象(字节串)，作为行为(字节串的结果，更为抽象)施动者或是作为对操作系统产生的效果(行为的结果，进一步抽象)的集合。反病毒技术就是按照"处理文件本身，根据文件处理事件，根据事件处理文件，处理环境本身"这个顺序逐渐发展的。

- 文件扫描：最早出现的反病毒软件都是将文件视为字节序列的。然而，研究者却很难将它称之为分析技术。因为它仅仅是将字节序列和已知特征做比较。研究者现在感兴趣的是这种技术的技术组件：在搜索恶意程序的过程中，数据被传递到判定组件，而该数据是从文件中抽取出的一个具有某种形式的有序字节序列。

  这种方法的特点是：反病毒软件只处理程序的字节码，而不关心它的行为。虽然方法比较传统，但是并没有过时，直到现在还被反病毒软件所采用，只是它已经不再是唯一的，甚至不再是主要的方法了，而是众多技术中的一种而已。
- 模拟：模拟技术从程度上来说处于把程序视为字节序列的技术和把程序视为行为序列的技术之间。

  虚拟机将程序字节代码划分为指令，并在虚拟的计算机环境中执行每一条指令。这样就可以监视程序的行为，不会像在真实环境中执行恶意程序那样会威胁操作系统和用户数据。

  虚拟机的特点主要是：虚拟机仍然和文件打交道，但是实际分析的对象已经是事件了。虚拟机已经用在了许多反病毒软件中，主要作为底层文件引擎或高层引擎(沙箱、系统监视)的一种辅助。
- 虚拟化：虚拟化使用的沙箱技术是模拟技术的一种逻辑延伸。使用沙箱技术时，恶意程序已经在真实的环境中运行了，只是被严格地控制起来了。模拟技术与虚拟化技术之间的界线可能并不宽，但却很明显。前一种技术为程序的执行提供了环境，这样工作进程中就包含了被执行程序并完全控制了程序的运行。对后一种技术，真

实的操作系统作为运行环境，而这种技术就是用来控制程序与操作系统之间交互的。这是两者的主要区别。基于虚拟技术的防御手段的处理对象已经不是文件，而是程序的行为，但还不是操作系统的行为。

反病毒软件不会主动使用“沙箱”以及虚拟机这类方法，主要是因为这些实现方法的程序需要占用大量的资源。带有沙箱的反病毒程序在程序和开始执行之间会有一个延时，所以可以通过这一点判断反病毒程序是否有“沙箱”。目前硬件虚拟化方面的研究正在积极进行之中，目前只有几种防病毒软件使用了“沙箱”引擎技术。

- 系统事件监控：系统事件监控是一种更为抽象的，为暴露恶意程序而进行信息搜集的方法。虚拟机或沙箱用于观察每一个独立的程序，监控器则是通过监视操作系统和运行的程序中产生的所有事件来观察所有的程序。

  这种信息搜集的方法是通过挂钩操作系统函数来实现的。挂钩了某些系统函数调用后，挂钩函数就能得到某个确定程序在系统中进行某个确定行为的信息。作为功能的延伸，监控器会收集这些行为的统计信息并将它传给分析组件进行处理。

  这种技术目前发展最为迅速。在某些较好的反病毒软件中，它已经成为一种组件，一个独立的工具，专门用来监控系统(如：Prevx、CyberHawk 这类的程序)。由于任何防御都可以被绕过，所以这种检测恶意程序的方法并非是最有效的，在真实环境中运行程序所带来的威胁会大大降低防御的效果。

- 搜索系统异常：搜索系统异常是搜集可能被感染的系统中信息的最为抽象的方法，它是前面技术的一种自然延伸与抽象。搜索系统异常方法建立在以下前提之上：操作系统和运行于其上的程序是一个完整的系统；操作系统有内在的“系统状态”；如果在操作系统中执行了恶意代码，则系统的状态就是“非健康的”，这种状态和系统中没有恶意代码时的“健康状态”不同。从这些状态入手，将系统状态与标准状态做对比或单独分析状态参数，研究者就能评价系统的状况，从而判断系统中是否可能有恶意程序。

为了有效采用分析异常的方法来检测恶意代码，分析系统必须足够强大，需要采用专家系统或神经网络技术。但同时也会面临以下一些问题：如何定义“健康状态”，它和“非健康状态”的区别，有哪些离散的参数可供跟踪以及如何进行分析等。由于技术的复杂性，因此目前这种方法用得还很少。某些 anti-rootkit 工具采用了这种方法，要么将系统状态和标准系统状态做对比(如 PatchFinder、Kaspersky Inspector)，要么比较单独的参数(如 GMER，Rootkit Unhooker)。

(2) 分析组件

分析组件包括对目标的简单比较、复杂比较或基于复杂数据分析的专家系统。病毒判定算法的复杂度是很难精确界定的。反病毒软件的分析系统大致可以分成三种，而在这三者中间可能还有许多中间变体。

- 简单比较：通过将单一对象与已有样本做比较而得出结论。比较的结果只有“是”或者“不是”。如采用严格确定的字符序列去识别恶意程序；或者通过行为比较发现可疑程序行为，如向注册表关键部位或自启动文件夹中写入的行为等。
- 复合比较：将一个或几个对象与相应样本做比较而得出结果。比较的模式可以是可变的，而比较的结果是一个概率。例如，经过对数据样本分析，判断 API A、B、C、

D 为恶意代码经常调用的 API。某可执行文件调用其中任意三个 API，则该可执行文件为恶意代码的概率大于 90%。

- 专家系统：通过对数据进行复杂的分析而得出结果。这是自身带有人工智能性质的系统。例如不使用严格给定的一组参数，而是从整体上多方面评估所有参数，考虑它们潜在恶意性的权重并计算出总体结果。

3. 恶意代码检查数据类型分析

检测数据可以分为静态和动态两种类型。针对两种类型数据的具体特点，将恶意代码检测技术分成基于代码序列特征码的检测技术、启发式的检测技术、基于异常行为的检测技术和基于行为结果的检测技术。

(1) 基于代码序列特征码的检测技术：基于代码序列特征码的检测是应用历史最长的一种方法。通过分析恶意代码样本，从样本的代码中提取它们的特征代码，在扫描文件时将当前的文件与特征码库进行对比，判断文件中是否含有特征数据，如有则认为是恶意代码。这种方法速度快，较为准确。但是也有明显的缺点：需要先捕获恶意代码才能分析掌握特征码，因此检测总是落后于恶意代码；另外对于采用了代码变形技术、代码混淆技术、代码加密技术及加壳技术等恶意代码自我保护技术的恶意代码，这种检测方法会失效。

(2) 启发式的检测技术：恶意代码要实现其特定功能，需要使用系统的 API 函数。通过静态分析和动态分析获得恶意代码调用的函数列表和函数间的调用关系。通过对大量样本的分析，将获得的函数列表和函数间的调用关系采用数据挖掘等方法进行分析，总结出特征并对每种调用关系定义一种危险级别。如果某个程序调用了危险的特定函数集合，研究者可以怀疑其可能是恶意代码。

(3) 基于异常行为的检测技术：利用病毒的特有行为特征来监测病毒的方法。当程序运行时，实时监视其运行过程中的动态行为，如果发现了恶意行为，立即报警并阻止其运行。目前较为流行的主动防御技术就是分析、监控系统内进程或程序的行为和指令，如果发现异常则立刻阻止恶意代码的执行以保护系统。

(4) 基于行为结果的检测技术：这种技术的原理是检测恶意代码执行后对系统状态的改变结果，然后进行判定。具体方法是建立不同操作系统版本和不同应用软件安装后的原始状态库。因为恶意代码执行后，将会使系统状态发生改变，如注册表、文件、进程以及端口等，所以通过分析系统状态的前后变化，可得出是否感染恶意代码的结论。

4. 现有检测技术的缺陷及未来发展趋势

至今还没有一项检测技术能够检测所有的恶意代码，并且误报、误杀成为反病毒行业面临的难题。随着恶意代码技术的不断提高，已有的检测技术都存在以下不足：误报率高，漏报率高，占用资源多，效率低，有效期短及防攻击能力弱等。因此，针对恶意代码新的检测和防范技术成为当前的研究热点。

随着恶意代码技术的发展，恶意代码呈现出以下特点：隐蔽性不断增强，传播方式和途径增多，网络化趋势明显，危害性增大等。计算机病毒、蠕虫和木马技术互相融合的趋势日益明显。这些情况使得恶意代码的检测和预防变得更加困难。因此，恶意代码复合检测技术是提高检测能力的重要途径之一。复合检测技术是指将现有的恶意代码检测技术进行综

合应用，同时与IDS和防火墙技术有机结合，在多个主机和网络等环节进行检测，才能适应恶意代码的发展，有效抵御恶意代码的侵袭。另一方面，近年来恶意代码制作工具化，导致恶意代码变种多，数量急剧增长，传统的特征代码检测技术已经无法适应恶意代码的这种变化形势。因此，基于数据挖掘、人工神经网络、决策树、最近邻居法、支持向量机和贝叶斯理论等机器学习的检测技术成为研究的热点，并逐步被广泛使用。尤其是基于系统和网络异常行为的恶意代码检测技术成为近期的主要发展趋势，基于可信计算技术的恶意代码主动防御技术是今后恶意代码检测和防范技术研究的重要发展方向。

14.2.4 开源恶意代码检测系统 Clam AntiVirus

1. 简介

Clam AntiVirus是一款UNIX下开源的(GPL)反病毒工具包，该工具包提供了包括灵活、可伸缩的监控程序、命令行扫描程序以及用于自动更新数据库的高级工具在内的大量实用程序。该工具包的核心在于可用于各类场合的反病毒引擎共享库。

Clam AntiVirus的主要特点如下：

(1) Clam AntiVirus内置了对包含Zip、RAR、Tar、Gzip、Bzip2、OLE2、Cabinet、CHM、BinHex、SIS及其他格式在内的多种压缩包格式的支持。

(2) 内置了对绝大多数邮件文件格式的支持。

(3) 内置了对使用UPX、FSG、Petite、NsPack、wwpack32、MEW、Upack压缩以及使用SUE、Y0da Cryptor和其他程序模糊处理的ELF可执行文件和便携式可执行文件的支持。

(4) 内置了对包括MS Office和MacOffice文件、HTML、RTF和PDF在内的主流文档格式的支持。

2. Clam AntiVirus的特征码格式

Clam AntiVirus主要采用特征匹配方式进行病毒的识别，其采用的特征码主要有：

(1) 文件MD5码：这种特征码是最简单的Clam AntiVirus特征码形式，但只能识别静态、单一的恶意代码。特征码的形式如下：

```
48c4533230e1ae1c118c741c0db19dfb:17387:test.exe
```

(2) PE区段MD5码：这种格式是将PE文件某个区段内容的MD5码作为Clam AntiVirus特征码。特征码的形式如下：

```
PESectionSize:MD5:MalwareName
```

(3) 十六进制特征码分为以下四种：

- 十六进制基本特征码格式，该格式比较简单，只需要设定恶意代码名称和应当包含的十六进制特征串。形式如下：

```
MalwareName=HexSignature
```

- 十六进制扩展特征码格式：在基本格式基础上加入了文件类型、偏移量等属性，如下：

```
MalwareName:TargetType:Offset:HexSignature[:MinEngineFunctionalityLevel:[Max]]
```

- 十六进制逻辑特征码格式：最为复杂的一种特征码格式，该格式可以支持多特征串以及它们之间复杂的逻辑关系，其格式定义如下：

```
SignatureName;TargetDescriptionBlock;LogicalExpression;Subsig0;Subsig1;Subsig2;
```

其中，TargetDescriptionBlock 属性规定了目标文件对应的扫描引擎类型；LogicalExpression 指定了特征码 Subsig0、Subsig1 和 Subsig2 之间的逻辑关系。举例说明如下：

```
Sig4;Target:1,Engine:18-20;((0|1)&(2|3))&4;EP+123:33c06834f04100f2aef7d14951684cf04100e811
0a00;S2+78:22??232c2d252229{-15}6e6573(63|64)61706528;S+50:68efa311c3b9963cb1ee8e58
6d32aeb9043e;f9c58dcf43987e4f519d629b103375;SL+550:6300680065005c0046006900
```

- 压缩文件特征码：此特征码主要针对各种压缩包内的文件，其格式如下：

```
virname:encrypted:filename:normal size:csize:crc32:cmethod:fileno:max depth
```

其中各属性的含义如下：

a. virusname：病毒名称；

b. encrypted：加密标记（1—加密；0—未加密）；

c. filename：文件名称；

d. normal size：原文件大小；

e. csize：压缩后的大小；

f. CRC32：CRC32 校验值；

g. Cmethod：压缩类型(.zip or .rar)；

h. Fileno：文件在压缩包中的序号；

i. Max depth：压缩包的最大层数。

3. BM 特征码匹配算法

如果将二进制文件看成连续字符序列，特征码匹配可以使用字符串匹配算法实现。Boyer 和 Moore 提出的 BM(Boyer-Moore)算法被认为在一般的应用中为最有效的字符串匹配算法。设待检测目标字符串长度为 m，模式匹配字符串长度为 n，这种算法的最好时间复杂度是 $O(n/m)$，最差复杂度是 $O(m\times n)$。

BM 算法的基本思想是：假设待检测目标字符串为 T，匹配参考字符串为 P，将 S 中自位置 i 起从右向左的一个子串与模式进行从右到左的匹配过程中，若发现不匹配，则下次应从 S 的 i + dist(S[i])位置开始重新进行新一轮的匹配，其效果相当于把 P 和 S 向右滑过一段距离 dist(S[i])，即跳过 dist(S[i])个无需进行比较的字符。

其中 dist(x)为距离函数，输入为一个无符号 char，输出为位置值，在该函数中维护一个长度为 256 的数组。由于无符号 char 的数值范围是 0～255，所以该数组每位对应一个无符号 char 在 P 中最后一次出现的位置距离字符串尾的长度，如 P 中不包含该无符号数，则数组该项为 P 的长度。这样，在匹配过程中，每次出现不匹配情况，可以直接跳过不必比较的

字符，从而提升比较效率。

下面举例说明，图 14-4 是使用 BM 算法进行字符串搜索的过程示意图。

图 14-4 为在字符串“ababcabcacbab”中寻找子串“abcac”的例子，寻找分为三步进行：第一步，指针偏移到 4 位置，然后开始回缩比对，对比的第一个字符“c”一致，发现第二个字符“b”与“a”不匹配；第二步，由于第三个字符“b”在“abcac”中的最后一次出现的位置距离字符串尾的距离为 3，则指针在原字符串中移动三位，从位置 3 到位置 6；第三步，由于第二步移动后，其第一个字符就不匹配，所以指针继续移动，依然移动 3 位，从位置 6 到达位置 9，继续回缩匹配，成功匹配后返回。

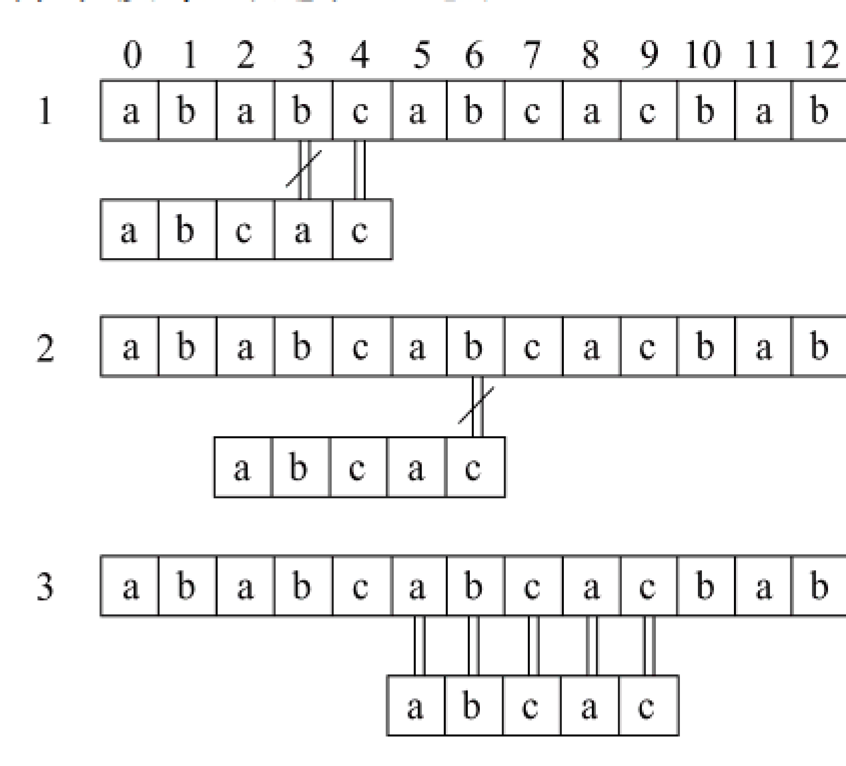


图 14-4 BM 算法匹配示意图

4. AC 特征码匹配算法

自动机匹配算法 AC(Aho-Corasick)算法是最著名的多模匹配算法之一。该算法在 1975 年产生于贝尔实验室。Aho-Corasick 算法是基于有限状态自动机进行特征码匹配的，在进行匹配之前，先对模式串集合进行预处理，构建树型有限状态自动机 FSA(finite state automata)，然后依据该 FSA，只需对文本串 T 扫描一次就可以找出与其匹配的所有模式串。

预处理生成 3 个函数：

(1) 转移函数用于标识，在当前状态下读入下一个待比较文本的字符后到达的下一个状态。

(2) 失效函数用来指明在某个状态下，当读入的字符不匹配时应转移到的下一个状态。

(3) 输出函数的作用是，在匹配过程中，当出现匹配时输出与当前状态匹配的模式。

Aho-Corasick 算法的匹配过程是：从初始状态 0 出发，每次取出文本串中的一个字符，根据当前的状态和扫描到的字符，利用转移函数或失效函数进入下一状态。当某个状态的输出函数不为空时，表明达到该状态时找到了匹配的模式，于是输出匹配结果。有限自动机的构造过程是将匹配关键字集合变换成由转向函数、失效函数和输出函数所组成的树型有限自动机。模式匹配的处理过程就变成了状态转换的处理过程，举例说明，由关键字集合{he，she，his，hers}构成的关键字模式树如图 14-5 所示。

在此基础上，补全失效函数和转移函数即可以基于{he，she，his，hers }关键词集合构造自动机。如图 14-6 所示，其中虚线为失效函数，没有给出的都指向根节点，实线为转移函数。在有限自动机的构造中，每个模式串的字符是从前到后依次加到树型的有限自动机中的，在匹配时，目标串的输入，即匹配过程，也是按照从前到后的次序尝试匹配。

利用已构成的有限自动机进行多串一遍查找的过程如下：

(1) 从有限自动机的 0 状态出发，从目标字符串的第一个字符开始正向逐个取出字符，并检测其转移函数或者失效函数进而进入下一个状态。

(2) 不断重复步骤(1)，直到整个字符串处理完毕后当前状态对应的输出函数不为空时为止。函数结束后输出匹配结果。

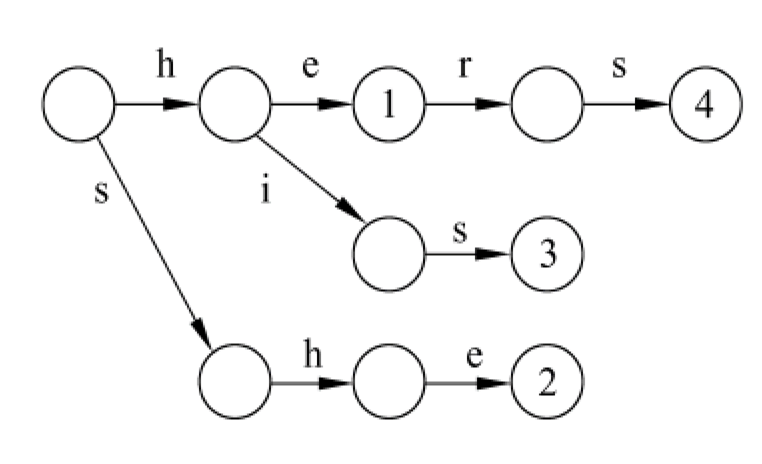


图 14-5 AC 算法模式树

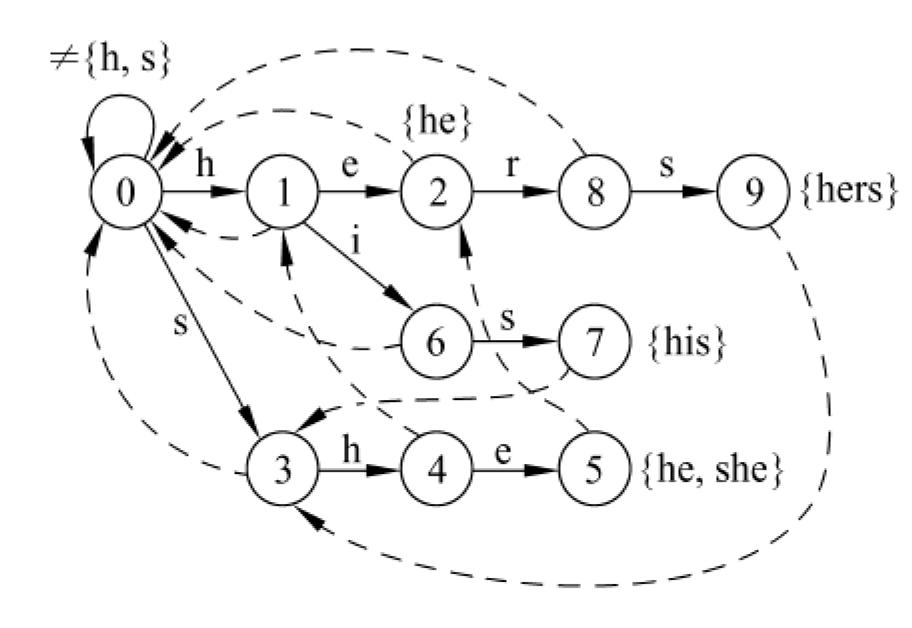


图 14-6 模式树构造

AC 算法可以认为是 KMP 算法在多串查找的扩展。其模式匹配的时间复杂度是 $O(n)$，而且与模式集中模式串的个数和每个模式串的长度无关。无论模式串 P 是否出现在待检测字符串 T 中，T 中的每个字符都必须输入状态机中。所以，无论是在最好情况还是最坏情况下，Aho-Corasick 算法模式匹配的时间复杂度都是 $O(n)$。包括预处理时间在内，Aho-Corasick 算法总时间复杂度是 $O(m+n)$，其中，m 为所有模式串 P 的长度总和。

14.3 实例编程练习

14.3.1 编程练习要求

本章要求在 Linux 平台上以 Clam AntiVirus 引擎和 Clam AntiVirus 特征码库为基础，完成 ClamScan 程序的设计、实现。由于病毒扫描程序的复杂度较高，这里仅对 Clam AntiVirus 的关键子程序进行分析和介绍，读者可以在此基础上自行编写属于自己的病毒扫描程序。下面给出编写 ClamScan 程序的具体要求。

ClamScan 是 Linux 命令行程序，命令行格式如下：

```
./ClamScan [输入文件路径]
```

[输入文件路径]指需要进行病毒扫描的文件路径。扫描结束后，将返回扫描结果如下：

```
root@ localhost:/tmp$clamscan malware.zip
malware.zip: Worm.Mydoom.U FOUND
-----------SCAN SUMMARY-----------
Known viruses: 1
Scanned directories: 0
Engine version: 0.92.1
Scanned files: 1
Infected files: 1
Data scanned: 0.02 MB
Time: 0.024 sec(0 m 0 s)
```

14.3.2 编程训练设计与分析

ClamScan 程序负责完成恶意代码检测工作，检测流程如图 14-7 所示。

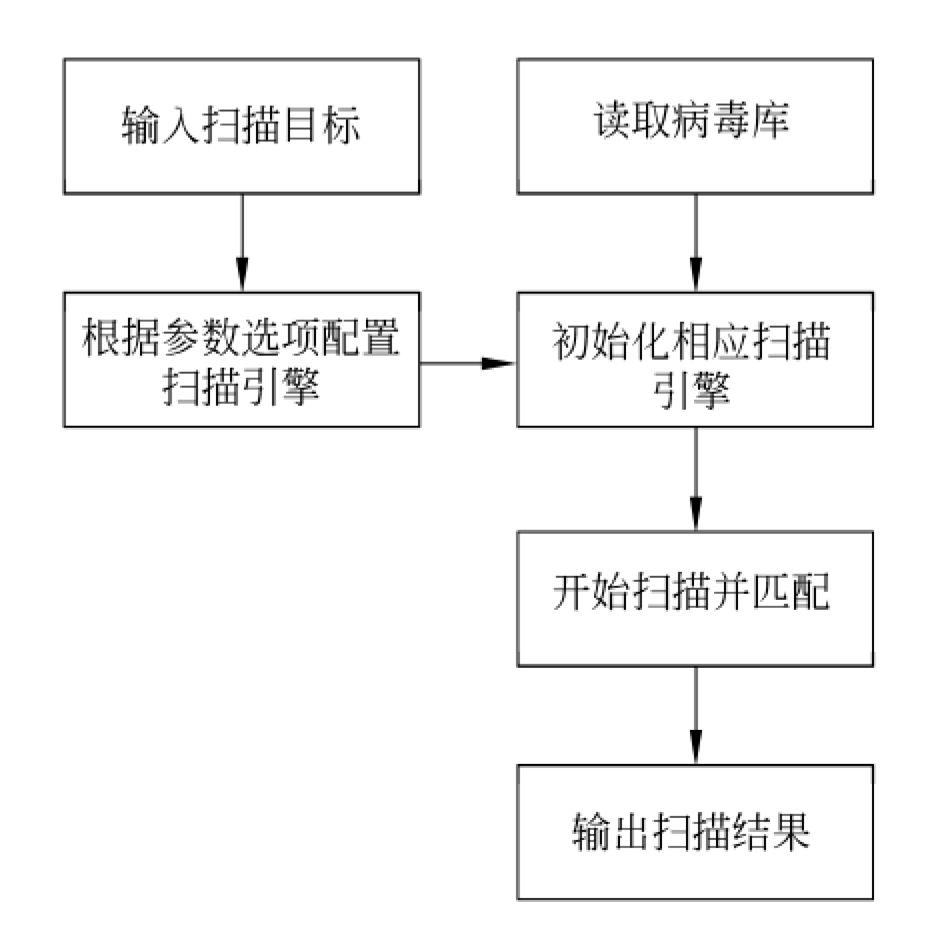


图 14-7 ClamScan 恶意代码检测流程图

1. 相关数据结构

头文件 ClamScan.h 中定义了若干重要的数据结构：主要用于定义扫描器结构，匹配器结构，不同算法参数等。

```
//扫描器结构
typedef struct {
    const char **virname;                        //病毒名
    unsigned long int * scanned;                 //已扫描的 Block 数 1Block=16KB
    const struct cli_matcher * root;             //匹配器
    const struct cl_engine * engine;             //扫描引擎
    unsigned long scansize;
    unsigned int options;                        //扫描选项
    unsigned int recursion;                      //对压缩文件的递归次数
    unsigned int scannedfiles;                   //扫描的文件数量
    unsigned int found_possibly_unwanted;
    struct cli_dconf * dconf;                    //文件类型设置
} cli_ctx;


struct cl_engine {
    uint32_t refcount;//reference counter
    uint32_t sdb;
    uint32_t dboptions;
    uint32_t dbversion[2];
    uint32_t ac_only;
    uint32_t ac_mindepth;                        //Aho-Corasick 算法 trie 树的最小深度
    uint32_t ac_maxdepth;                        //Aho-Corasick 算法 trie 树的最大深度
    char * tmpdir;                               //病毒库临时释放文件
    uint32_t keeptmp;
```

```
    //Limits
    uint64_t maxscansize;                   //在扫描压缩文件时,压缩文件的最大文件大小
    uint64_t maxfilesize;                   //压缩包中,可以扫描文件的最大体积
    uint32_t maxreclevel;                   //可扫描压缩文件的最大压缩层数
    uint32_t maxfiles;                      //可扫描压缩包中最大的文件数量

    uint32_t min_cc_count;
    uint32_t min_ssn_count;
    //匹配器
    struct cli_matcher** root;
    //针对 MD5 特征代码类型的 B-M 算法匹配器
    struct cli_matcher * md5_hdb;
    //针对 PE 段 MD5 特征码类型的 B-M 算法匹配器
    struct cli_matcher * md5_mdb;
    //针对白名单库的 B-M 算法匹配器
    struct cli_matcher * md5_fp;
    //Zip 文件
    struct cli_meta_node * zip_mlist;
    //RAR
    struct cli_meta_node * rar_mlist;
    //针对 Phishing .pdb and .wdb 特征库正则表达式匹配器
    struct regex_matcher * whitelist_matcher;
    struct regex_matcher * domainlist_matcher;
    struct phishcheck * phishcheck;
    //动态配置
    struct cli_dconf * dconf;
    //文件类型定义
    struct cli_ftype * ftypes;
    //可被忽略的特征代码
    struct cli_ignored * ignored;
    //PUA 类
    char * pua_cats;
    //内存池
    mpool_t * mempool;
};
struct cl_settings {
    uint32_t ac_only;                       //只用 Aho-Corasick 算法
    uint32_t ac_mindepth;                   //Aho-Corasick 算法最小深度
    uint32_t ac_maxdepth;                   //Aho-Corasick 算法最大深度
    char * tmpdir;                          //病毒库
    uint32_t keeptmp;                       //释放文件
    uint64_t maxscansize;                   //在扫描压缩文件时,压缩文件的最大文件大小
    uint64_t maxfilesize;                   //压缩包中,可以扫描文件的最大体积
    uint32_t maxreclevel;                   //可扫描压缩文件的最大压缩层数
    uint32_t maxfiles;                      //可扫描压缩包中最大的文件数量
```

```
        uint32_t min_cc_count;
        uint32_t min_ssn_count;
    char * pua_cats;
    }
    //匹配器数据结构
    struct cli_matcher {
        //扩展的 B-M匹配算法
        uint8_t * bm_shift;                    //B-M算法的移动位数
        struct cli_bm_patt**bm_suffix;         //B-M算法的特征值后缀(好后缀原则)
        struct hashset md5_sizes_hs;
        uint32_t * soff, soff_len;             //PE文件的区段偏移量和长度
        uint32_t bm_patterns;                  //B-M算法的特征值的数量
        //扩展的 Aho-Corasick 匹配算法相关变量
    uint32_t ac_partsigs, ac_nodes, ac_patterns, ac_lsigs;
    //ac_partsigs : A-C算法的部分匹配特征值数量;
    //ac_nodes: A-C算法 trie树的节点;
    //ac_patterns: A-C算法的部分特征值数量;
    //ac_lsigs: A-C算法的逻辑特征代码数量
    struct cli_ac_lsig**ac_lsigtable;             //A-C算法逻辑表
    //* ac_root: A-C算法 trie树的根节点;
    // **ac_nodetable: A-C算法 trie树的节点表

        struct cli_ac_node * ac_root,**ac_nodetable;
        struct cli_ac_patt**ac_pattable;       //A-C算法特征值表
        uint8_t ac_mindepth, ac_maxdepth;      //A-C算法 Trie树的最小深度,最大深度

        uint16_t maxpatlen;                    //最大特征串长度
        uint8_t ac_only;                       //是否只用 A-C算法进行匹配。0:可以,1:不可以
    #ifdef USE_MPOOL
        mpool_t * mempool;
    #endif
    //B-M算法的特征串节点数据结构
    struct cli_bm_patt {
        unsigned char * pattern, * prefix;     //特征串,前缀
        char * virname, * offset;              //病毒名称,特征值对应的偏移量
        struct cli_bm_patt * next;             //B-M算法特征链表中下一个对象节点
        uint16_t length, prefix_length;        //特征串长度,前缀长度
        uint16_t cnt;                          //计数器
        unsigned char pattern0;                //特征串第一个字符
        uint8_t target;
    };
    //A-C算法的特征串节点数据结构
    struct cli_ac_patt {
        //特征串,前缀,特征串长度,前缀长度
```

```
uint16_t * pattern, * prefix, length, prefix_length;
    uint32_t mindist, maxdist;                    //最短距离,最长距离
    uint32_t sigid;                               //特征子串的标号
    uint32_t lsigid[3];                           //逻辑特征子串的标号
    uint16_t ch[2];
    char * virname, * offset;                     //病毒名称,偏移量
    void * customdata;                            //自定义数据
    uint16_t ch_mindist[2];                       //最小距离
    uint16_t ch_maxdist[2];
    uint16_t parts, partno, alt, alt_pattern;
    struct cli_ac_alt**alttable;                  //转移表
    struct cli_ac_patt * next, * next_same;
    uint8_t depth;
    uint16_t rtype, type;
};
```

2. 病毒库导入

为了实现扫描工作,程序必须在初始化之前读取、加载病毒库。该工作由函数 cli_loaddb 负责完成。函数通过 FILE * 文件指针读取病毒库文件,并按照其相关格式将病毒库信息载入内存。

```
static int cli_loaddb(FILE * fs, struct cl_engine * engine, unsigned int * signo, unsigned int options,
struct cli_dbio * dbio, const char * dbname)
{
    char buffer[FILEBUFF], * pt, * start;
    unsigned int line=0, sigs=0;
    int ret=0;
    struct cli_matcher * root;
    //初始化匹配器根节点
    if((ret=cli_initroots(engine, options)))
    return ret;
    root=engine->root[0];                        //获取根节点
    while(cli_dbgets(buffer, FILEBUFF, fs, dbio)) {
    line++;
    cli_chomp(buffer);
    pt=strchr(buffer, '=');
    if(!pt) {
        cli_errmsg("Malformed pattern line %d\n", line);
        ret=CL_EMALFDB;
        break;
    }
    start=buffer;
    * pt++=0;
    if(engine->ignored && cli_chkign(engine->ignored, dbname, line, start))
```

```
            continue;
        if(*pt=='=')continue;
    //将获得的特征值节点加入到根节点所在链表中
        if((ret=cli_parse_add(root, start, pt, 0, 0, NULL, 0, NULL, options))) {
            ret=CL_EMALFDB;
            break;
        }
        sigs++;
        }
        if(!line) {
        cli_errmsg("Empty database file\n");
        return CL_EMALFDB;
        }
        if(ret) {
        cli_errmsg("Problem parsing database at line %d\n", line);
        return ret;
        }
        if(signo)
         *signo+=sigs;
        return CL_SUCCESS;
    }
```

函数 cli_parse_add 用于在程序内存中特征值库里添加新的特征值信息。在导入病毒库的过程中，系统解析病毒库文件，从中提取特征值，并通过调用该函数将特征值导入。特征值添加函数的实现如下：

```
    int cli_parse_add(struct cli_matcher *root, const char *virname,const char *hexsig, uint16_t rtype,
    uint16_t type,const char *offset, uint8_t target, const uint32_t *lsigid,unsigned int options)
    {
        struct cli_bm_patt *bm_new;
        char *pt, *hexcpy, *start, *n;
        int ret, asterisk=0;
        unsigned int i, j, len, parts=0;
        int mindist=0, maxdist=0, error=0;

        if(strchr(hexsig, '{')) {
        root->ac_partsigs++;
        if(!(hexcpy=cli_strdup(hexsig)))
            return CL_EMEM;
        len=strlen(hexsig);
        for(i=0; i<len; i++)
            if(hexsig[i]=='{'||hexsig[i]=='*')
            parts++;
        if(parts)
            parts++;
        start=pt=hexcpy;
```

```
    for(i=1; i<=parts; i++) {
        if(i !=parts) {
        for(j=0; j<strlen(start); j++) {
            if(start[j]=='{') {
            asterisk=0;
            pt=start+j;
            break;
            }
            if(start[j]=='*') {
            asterisk=1;
            pt=start+j;
            break;
            }
        }
     *pt++=0;
        }
//增加 A-C 匹配算法的特征值
        if((ret=cli_ac_addsig(root, virname, start, root->ac_partsigs, parts, i, rtype, type,
mindist, maxdist, offset, lsigid, options))) {
        cli_errmsg("cli_parse_add(): Problem adding signature (1).\n");
        error=1;
        break;
        }
        if(i==parts)
        break;
        mindist=maxdist=0;
        if(asterisk) {
        start=pt;
        continue;
        }
        if(!(start=strchr(pt, '}'))) {
        error=1;
        break;
        }
        *start++=0;
        if(!pt) {
        error=1;
        break;
        }
        if(!strchr(pt, '-')) {
        if(!cli_isnumber(pt) || (mindist=maxdist=atoi(pt))<0) {
            error=1;
            break;
        }
        } else {
```

```
            if((n=cli_strtok(pt, 0, "-"))){
                if(!cli_isnumber(n) || (mindist=atoi(n))<0) {
                error=1;
                free(n);
                break;
                }
                free(n);
            }
            if((n=cli_strtok(pt, 1, "-"))) {
                if(!cli_isnumber(n) || (maxdist=atoi(n))<0) {
                error=1;
                free(n);
                break;
                }
                free(n);
            }
            if((n=cli_strtok(pt, 2, "-"))){//strict check
                error=1;
                free(n);
                break;
            }
            }
        }
        free(hexcpy);
        if(error)
            return CL_EMALFDB;
        } else if(strchr(hexsig, '*')) {
        root->ac_partsigs++;
        len=strlen(hexsig);
        for(i=0; i<len; i++)
            if(hexsig[i]=='*')
            parts++;
        if(parts)
            parts++;
        for(i=1; i<=parts; i++) {
            if((pt=cli_strtok(hexsig, i-1, "*"))==NULL) {
            cli_errmsg("Can't extract part %d of partial signature.\n", i);
            return CL_EMALFDB;
            }
            if((ret=cli_ac_addsig(root, virname, pt, root->ac_partsigs, parts, i, rtype, type, 0, 0,
            offset, lsigid, options))) {
            cli_errmsg("cli_parse_add(): Problem adding signature (2).\n");
            free(pt);
            return ret;
            }
```

```
            free(pt);
        }
        } else if(root->ac_only||strpbrk(hexsig, "? (")||type||lsigid) {
        if((ret=cli_ac_addsig(root, virname, hexsig, 0, 0, 0, rtype, type, 0, 0, offset, lsigid, options))) {
            cli_errmsg("cli_parse_add(): Problem adding signature (3).\n");
            return ret;
        }
    } else {
    //如果不是A-C算法的特征值或初始化A-C特征值节点失败,则初始化B-M算法的特征值节点
        bm_new=(struct cli_bm_patt*) mpool_calloc(root->mempool, 1,sizeof(struct cli_bm_patt));
        if(!bm_new)
            return CL_EMEM;
            bm_new->pattern=(unsigned char*) cli_mpool_hex2str(root->mempool, hexsig);
        if(!bm_new->pattern) {
            mpool_free(root->mempool, bm_new);
            return CL_EMALFDB;
        }
        bm_new->length=strlen(hexsig)/2;
        bm_new->virname=cli_mpool_virname(root->mempool,(char*) virname, options & CL_DB_OFFICIAL);
        if(!bm_new->virname) {
            mpool_free(root->mempool, bm_new->pattern);
            mpool_free(root->mempool, bm_new);
            return CL_EMEM;
        }
        if(offset) {
            bm_new->offset=cli_mpool_strdup(root->mempool, offset);
            if(!bm_new->offset) {
                mpool_free(root->mempool, bm_new->pattern);
            mpool_free(root->mempool, bm_new->virname);
            mpool_free(root->mempool, bm_new);
            return CL_EMEM;
            }
        }
        bm_new->target=target;
        if(bm_new->length>root->maxpatlen)
            root->maxpatlen=bm_new->length;
        //将特征值节点加入到B-M匹配算法的特征值链表中
        if((ret=cli_bm_addpatt(root, bm_new))) {
            cli_errmsg("cli_parse_add(): Problem adding signature(4).\n");
            mpool_free(root->mempool, bm_new->pattern);
            mpool_free(root->mempool, bm_new->virname);
            mpool_free(root->mempool, bm_new);
            return ret;
        }
    }
```

```
return CL_SUCCESS;
```

3. AC 特征码匹配算法初始化

AC 特征码匹配算法的初始化工作由函数 cli_ac_initdata 完成。函数工作非常简单，主要是从事分配相关内存，完成参数赋值等工作。

```
int cli_ac_initdata(struct cli_ac_data * data, uint32_t partsigs, uint32_t lsigs,uint8_t tracklen)
{
    unsigned int i;

    if(!data) {
    cli_errmsg("cli_ac_init: data==NULL\n");
    return CL_ENULLARG;
    }
    data->partsigs=partsigs;
    if(partsigs) {
    data->offmatrix= (int32_t***) cli_calloc(partsigs, sizeof(int32_t**));
    if(!data->offmatrix) {
        cli_errmsg("cli_ac_init: Can't allocate memory for data->offmatrix\n");
        return CL_EMEM;
    }
    }

    data->lsigs=lsigs;
    if(lsigs) {
    data->lsigcnt= (uint32_t**) cli_malloc(lsigs * sizeof(uint32_t * ));
    if(!data->lsigcnt) {
        if(partsigs)
        free(data->offmatrix);
        cli_errmsg("cli_ac_init: Can't allocate memory for data->lsigcnt\n");
        return CL_EMEM;
    }
    data->lsigcnt[0]= (uint32_t * ) cli_calloc(lsigs * 64, sizeof(uint32_t));
    if(!data->lsigcnt[0]) {
        free(data->lsigcnt);
        if(partsigs)
        free(data->offmatrix);
        cli_errmsg("cli_ac_init: Can't allocate memory for data->lsigcnt[0]\n");
        return CL_EMEM;
    }
    for(i=1; i<lsigs; i++)
        data->lsigcnt[i]=data->lsigcnt[0]+64 * i;
     }

    return CL_SUCCESS;
```

```
}
```

4. AC 特征码匹配算法匹配查找

AC 特征码匹配算法用于在一段指定的内存字节串中寻找符合条件的子串。其使用的算法即为第 14.2 节介绍的 Aho-Corasick 模式匹配算法。程序通过调用该函数实现对程序中特征码的定位。

```
inline static int ac_findmatch(const unsigned char * buffer, uint32_t offset, uint32_t length, const
struct cli_ac_patt * pattern, uint32_t * end)
{
    uint32_t bp, match;
    uint16_t wc, i, j, altcnt=pattern->alt_pattern;
    struct cli_ac_alt * alt;

    if((offset+pattern->length > length) || (pattern->prefix_length> offset))
    return 0;
    bp=offset+pattern->depth;
    match=1;
    for(i=pattern->depth; i<pattern->length && bp< length; i++) {
    AC_MATCH_CHAR(pattern->pattern[i],buffer[bp]);
    if(!match)
        return 0;
    bp++;
    }
    * end=bp;
    if(!(pattern->ch[1] & CLI_MATCH_IGNORE)) {
    bp +=pattern->ch_mindist[1];
    for(i=pattern->ch_mindist[1]; i<=pattern->ch_maxdist[1]; i++) {
        if(bp >= length)
        return 0;
        match=1;
        AC_MATCH_CHAR(pattern->ch[1],buffer[bp]);
        if(match)
        break;
        bp++;
    }
    if(!match)
        return 0;
    }
    if(pattern->prefix) {
    altcnt=0;
    bp=offset-pattern->prefix_length;
    match=1;
    for(i=0; i<pattern->prefix_length; i++) {
```

```
            AC_MATCH_CHAR(pattern->prefix[i],buffer[bp]);
            if(!match)
            return 0;
            bp++;
        }
        }
        if(!(pattern->ch[0] & CLI_MATCH_IGNORE)) {
        bp=offset-pattern->prefix_length;
        if(pattern->ch_mindist[0]+(uint32_t) 1>bp)
            return 0;
        bp-=pattern->ch_mindist[0]+1;
        for(i=pattern->ch_mindist[0]; i<=pattern->ch_maxdist[0]; i++) {
            match=1;
            AC_MATCH_CHAR(pattern->ch[0],buffer[bp]);
            if(match)
            break;
            if(!bp)
            return 0;
            else
            bp--;
        }
        if(!match)
            return 0;
        }
        return 1;
}
```

5. A-C 算法的扫描匹配函数

函数 cli_ac_scanbuff 用于实现恶意代码的甄别。系统将待扫描文件读入内存,并将文件的内存映像通过第一个参数 buffer 传递给该函数,该函数遍历初始化时读入的若干特征值信息,并依次对该文件的内存映像进行匹配尝试,进而判断该文件究竟是否包含恶意代码,并将判断结果返回调用者。

```
int cli_ac_scanbuff(const unsigned char * buffer,uint32_t length,const char**virname,void**customdata,
struct cli_ac_result**res,const struct cli_matcher * root,struct cli_ac_data * mdata,uint32_t offset,cli
_file_t ftype,int fd,struct cli_matched_type**ftoffset,unsigned int mode,const cli_ctx * ctx)
{
    struct cli_ac_node * current;
    struct cli_ac_patt * patt, * pt;
        uint32_t i,bp,realoff,matchend;
    uint16_t j;
    int32_t**offmatrix;
    uint8_t found;
    struct cli_target_info info;
    int type=CL_CLEAN;
```

```
struct cli_ac_result * newres;

if(!root->ac_root)
return CL_CLEAN;
if(!mdata) {
cli_errmsg("cli_ac_scanbuff: mdata==NULL\n");
return CL_ENULLARG;
}
memset(&info,0,sizeof(info));
current=root->ac_root;
for(i=0;i<length;i++)  {
//当前节点是否是叶子节点,如果是则转向失败转移函数
if(IS_LEAF(current))
    current=current->fail;

current=current->trans[buffer[i]];
//从 A-C 自动机中获得最终的特征值
if(IS_FINAL(current)) {
    patt=current->list;
    while(patt) {
    bp=i+1-patt->depth;
//判断是否匹配
    if(ac_findmatch(buffer,bp,length,patt,&matchend)) {
        pt=patt;
        while(pt) {
        if((pt->type && !(mode & AC_SCAN_FT))||(!pt->type && !(mode & AC_SCAN_VIR))) {
            pt=pt->next_same;
            continue;
        }
        realoff=offset+bp-pt->prefix_length;
        if(pt->offset && (!pt->sigid||pt->partno==1)) {
            if(!cli_validatesig(ftype,pt->offset,realoff,&info,fd,pt->virname)) {
            pt=pt->next_same;
            continue;
            }
        }
  //如果存在 sigid,则 pt 是部分特征值
        if(pt->sigid) {
```

```
if(pt->partno!=1 && (!mdata->offmatrix[pt->sigid-1] || !mdata->offmatrix[pt->
sigid-1][pt->partno-2][0])) {
pt=pt->next_same;
continue;
}
if(!mdata->offmatrix[pt->sigid-1]) {
mdata->offmatrix[pt->sigid-1]=cli_malloc(pt->parts * sizeof(int32_t *));
if(!mdata->offmatrix[pt->sigid-1]) {
    cli_errmsg("cli_ac_scanbuff: Can't allocate memory for mdata->offmatrix[%u]\
    n",pt->sigid-1);
    if(info.exeinfo.section)
    free(info.exeinfo.section);
    return CL_EMEM;
}
 mdata->offmatrix[pt->sigid-1][0]=cli_malloc(pt->parts * (CLI_DEFAULT_AC_
 TRACKLEN+1) * sizeof(int32_t));
if(!mdata->offmatrix[pt->sigid-1][0]) {
    cli_errmsg("cli_ac_scanbuff: Can't allocate memory for mdata->offmatrix[%u][0]
    \n",pt->sigid-1);
    free(mdata->offmatrix[pt->sigid-1]);
    mdata->offmatrix[pt->sigid-1]=NULL;
    if(info.exeinfo.section)
    free(info.exeinfo.section);
    return CL_EMEM;
}
memset(mdata->offmatrix[pt->sigid-1][0],-1,pt->parts * (CLI_DEFAULT_AC_TRACKLEN
+1) * sizeof(int32_t));
mdata->offmatrix[pt->sigid-1][0][0]=0;
for(j=1;j<pt->parts;j++) {
    mdata->offmatrix[pt->sigid-1][j]=mdata->offmatrix[pt->sigid-1][0]+j *
    (CLI_DEFAULT_AC_TRACKLEN+1);
    mdata->offmatrix[pt->sigid-1][j][0]=0;
}
}
offmatrix=mdata->offmatrix[pt->sigid-1];
if(pt->partno !=1) {
found=0;
for(j=1;j<=CLI_DEFAULT_AC_TRACKLEN && offmatrix[pt->partno-2][j] !=-1;j++) {
    found=1;
    if(pt->maxdist)
    if(realoff-offmatrix[pt->partno-2][j] >pt->maxdist)
        found=0;
    if(found && pt->mindist)
    if(realoff-offmatrix[pt->partno-2][j]<pt->mindist)
        found=0;
```

```
            if(found)
            break;
        }
        }
        if(pt->partno==1||(found && (pt->partno !=pt->parts))) {
        offmatrix[pt->partno-1][0] %=CLI_DEFAULT_AC_TRACKLEN;
        offmatrix[pt->partno-1][0]++;
        offmatrix[pt->partno-1][offmatrix[pt->partno-1][0]]=offset+matchend;
        if(pt->partno==1)                //保存第一部分特征值的偏移量
            offmatrix[pt->parts-1][offmatrix[pt->partno-1][0]]=realoff;
        } else if(found && pt->partno==pt->parts) {
        //判断特征值中标称的类型
    if(pt->type) {
            if(pt->type==CL_TYPE_IGNORED && (!pt->rtype||ftype==pt->rtype)) {
            if(info.exeinfo.section)
                free(info.exeinfo.section);
            return CL_TYPE_IGNORED;
            }
            if((pt->type>type||pt->type>=CL_TYPE_SFX||pt->type==CL_TYPE_MSEXE) &&
            (!pt->rtype||ftype==pt->rtype)) {
            cli_dbgmsg("Matched signature for file type %s\n",pt->virname);
            type=pt->type;
            if(ftoffset && (!*ftoffset||(*ftoffset)->cnt<MAX_EMBEDDED_OBJ||type==CL_
            TYPE_ZIPSFX) && (type>=CL_TYPE_SFX||((ftype==CL_TYPE_MSEXE||ftype==CL_TYPE
            _ZIP||ftype==CL_TYPE_MSOLE2) && type==CL_TYPE_MSEXE))){
                for(j=1;j<=CLI_DEFAULT_AC_TRACKLEN && offmatrix[0][j] !=-1;j++) {
                if(ac_addtype(ftoffset,type,offmatrix[pt->parts-1][j],ctx)) {
                    if(info.exeinfo.section)
                    free(info.exeinfo.section);
                    return CL_EMEM;
                }
                }
            }
            memset(offmatrix[0],-1,pt->parts*(CLI_DEFAULT_AC_TRACKLEN+1)*sizeof(int32
            _t));
            for(j=0;j<pt->parts;j++)
                offmatrix[j][0]=0;
            }
        } else {                        //如果没有规定类型,进入逻辑特征值匹配状态
            if(pt->lsigid[0]) {
            mdata->lsigcnt[pt->lsigid[1]][pt->lsigid[2]]++;
            pt=pt->next;
            continue;
            }
//收集匹配结果
```

```
                if(res) {
                newres= (struct cli_ac_result * ) malloc(sizeof(struct cli_ac_result));
                if(!newres) {
                    if(info.exeinfo.section)
                    free(info.exeinfo.section);
                    return CL_EMEM;
                }
                newres->virname=pt->virname;
                newres->customdata=pt->customdata;
                newres->next= * res;
               * res=newres;
                pt=pt->next;
                continue;
                } else {
                if(virname)
                    * virname=pt->virname;
                if(customdata)
                    * customdata=pt->customdata;
                if(info.exeinfo.section)
                    free(info.exeinfo.section);
                return CL_VIRUS;
                }
            }
            }
        } else {  //如果是旧格式的特征值
            if(pt->type) {
            if(pt->type==CL_TYPE_IGNORED && (!pt->rtype||ftype==pt->rtype)) {
                if(info.exeinfo.section)
                free(info.exeinfo.section);
                return CL_TYPE_IGNORED;
            }
            if((pt->type>type||pt->type>=CL_TYPE_SFX||pt->type==CL_TYPE_MSEXE) && (!pt
            ->rtype||ftype==pt->rtype)) {
                cli_dbgmsg("Matched signature for file type %s at %u\n",pt->virname,realoff);
                type=pt->type;
                if(ftoffset && (! * ftoffset||(* ftoffset)->cnt<MAX_EMBEDDED_OBJ||type==CL_
                TYPE_ZIPSFX) && (type>=CL_TYPE_SFX||((ftype==CL_TYPE_MSEXE||ftype==CL_TYPE
                _ZIP||ftype==CL_TYPE_MSOLE2) && type==CL_TYPE_MSEXE))){
                if(ac_addtype(ftoffset,type,realoff,ctx)) {
                    if(info.exeinfo.section)
                    free(info.exeinfo.section);
                    return CL_EMEM;
                }
                }
            }
```

```
                } else {
                if(pt->lsigid[0]) {
                    mdata->lsigcnt[pt->lsigid[1]][pt->lsigid[2]]++;
                    pt=pt->next;
                    continue;
                }
                if(res) {
                    newres=(struct cli_ac_result*)malloc(sizeof(struct cli_ac_result));
                    if(!newres) {
                    if(info.exeinfo.section)
                        free(info.exeinfo.section);
                    return CL_EMEM;
                    }
                    newres->virname=pt->virname;
                    newres->customdata=pt->customdata;
                    newres->next=*res;
                   *res=newres;
                    pt=pt->next;
                    continue;
                } else {
                    if(virname)
                   *virname=pt->virname;
                    if(customdata)
                   *customdata=pt->customdata;
                    if(info.exeinfo.section)
                    free(info.exeinfo.section);
                    return CL_VIRUS;
                }
                }
            }
            pt=pt->next_same;
            }
        }
        patt=patt->next;
        }
    }
    }
    if(info.exeinfo.section)
    free(info.exeinfo.section);
    return(mode & AC_SCAN_FT) ? type : CL_CLEAN;
}
```

6. B-M 算法的扫描匹配函数

函数 cli_bm_scanbuff 用于使用 B-M 算法实现文件恶意代码甄别。该函数与函数

cli_ac_scanbuff 的功能基本相同，其主要区别是该函数使用 Boyer-Moore 算法实现特征值匹配运算实现特征值定位。

```
int cli_bm_scanbuff(const unsigned char * buffer, uint32_t length, const char**virname, const struct cli_
matcher * root, uint32_t offset, cli_file_t ftype, int fd)
{
    uint32_t i, j, off;
    uint8_t found, pchain, shift;
    uint16_t idx, idxchk;
    struct cli_bm_patt * p;
    const unsigned char * bp, * pt;
    unsigned char prefix;
    struct cli_target_info info;

    if(!root||!root->bm_shift)
    return CL_CLEAN;
    if(length<BM_MIN_LENGTH)
    return CL_CLEAN;
    memset(&info, 0, sizeof(info));
    for(i=BM_MIN_LENGTH-BM_BLOCK_SIZE; i<length-BM_BLOCK_SIZE+1; ) {
    idx=HASH(buffer[i], buffer[i+1], buffer[i+2]);
    shift=root->bm_shift[idx];
    if(shift==0) {
        //从缓冲区读取指定长度的字节，用于前缀比对
        prefix=buffer[i-BM_MIN_LENGTH+BM_BLOCK_SIZE];
        p=root->bm_suffix[idx];
//从 B-M算法特征值链表中取出 p进行比对
pchain=0;
        while(p) {
        if(p->pattern0!=prefix) {
            if(pchain)
            break;
            p=p->next;
            continue;
        } else pchain=1;
        off=i-BM_MIN_LENGTH+BM_BLOCK_SIZE;
        bp=buffer+off;

        if((off+p->length>length)||(p->prefix_length>off)) {
            p=p->next;
            continue;
        }
        idxchk=MIN(p->length, length-off)-1;
        if(idxchk) {
        //做一次转移后，查找是否匹配
```

```
        if((bp[idxchk] != p->pattern[idxchk]) || (bp[idxchk / 2] != p->pattern[idxchk / 2])) {
        p=p->next;
        continue;
        }
    }
    if(p->prefix_length) {
        off-=p->prefix_length;                    //转移长度为特征串前缀长度
        bp-=p->prefix_length;
        pt=p->prefix;
    } else {
        pt=p->pattern;
    }
    found=1;
    for(j=0; j<p->length+p->prefix_length && off<length; j++, off++) {
        if(bp[j]!=pt[j]) {
//判断是否匹配,并保存位置 j
        found=0;
        break;
        }
    }
    if(found && p->length+p->prefix_length==j) {
//如果完全匹配
        if(p->offset) {
//获得文件偏移量
        off=offset+i-p->prefix_length - (BM_MIN_LENGTH-BM_BLOCK_SIZE);
        if(!cli_validatesig(ftype, p->offset, off, &info, fd, p->virname)) {
//验证特征值,同时获得病毒名称
            p=p->next;
            continue;
        }
        }
        if(virname)
        * virname=p->virname;
        if(info.exeinfo.section)
        free(info.exeinfo.section);
        return CL_VIRUS;
    }
    p=p->next;
    }
    shift=1;
}
i+=shift;
}
if(info.exeinfo.section)
free(info.exeinfo.section);
```

```
    return CL_CLEAN;
}
```

14.4 扩展与提高

14.4.1 使用 Clam AntiVirus 扫描邮件

Clam AntiVirus 作为 Linux 平台下重要的病毒检测工具，得到了很多第三方软件的支持。下面介绍一种邮件代理网关 p3scan 与 Clam AntiVirus 的组合配置方法。

(1) 安装、编译 Clam AntiVirus 后，配置/etc/clamd.conf 文件如下：

```
LogFile/var/log/clamav/clamd.log
LogFileMaxSize 0
LogTime yes
PidFile/var/run/clamav/clamd.pid
TemporaryDirectory/dev/shm
DatabaseDirectory/var/lib/clamav
FixStaleSocket yes
TCPSocket 3310
TCPAddr 127.0.0.1
MaxConnectionQueueLength 80
StreamMaxLength 20M
MaxThreads 80
ReadTimeout 300
MaxDirectoryRecursion 20
User clamav
AllowSupplementaryGroups yes
DetectBrokenExecutables yes
ScanPDF yes
ArchiveMaxFileSize 64M
ArchiveMaxRecursion 10
ArchiveMaxCompressionRatio 300
ArchiveBlockEncrypted no
ArchiveBlockMax yes
ClamukoMaxFileSize 64M
```

(2) 启动 clamd：

```
/etc/rc.d/init.d/clamd start
```

(3) 安装、编译 p3scan 后，配置 p3scan

配置/etc/p3scan/p3scan.conf 如下：

```
user=root
scannertype=clamd
```

```
scanner=127.0.0.1:3310
virusregexp=.* : (.* ) FOUND
timeout=90
footer=/usr/bin/clamdscan-V
```

为原有邮件模板创建一个连接：

```
ln-s p3scan-en.mail p3scan.mail
```

(4) 启动 p3scan：

```
/etc/rc.d/init.d/p3scan start
```

(5) 设置 iptables：

设置 iptables 规则，把所有到 SMTP(25)，POP3(110)的流量重定向到 p3scan 的端口 8110 上：

```
iptables-t nat-A PREROUTING-p tcp-m multiport --dport 25,110-j REDIRECT--to-port 8110
```

14.4.2 基于可信计算技术的恶意代码主动防御技术

如果信息系统中每一个使用者都是经过认证和授权的，其操作都是符合安全要求的，那么就不会产生人为的攻击性事故，就能保证整个信息系统的安全。这是不需要证明的公理，是信息系统安全所追求的目标。

1. 当前信息安全系统存在的问题

当前，大部分信息安全系统主要由防火墙、入侵检测和病毒防范等组成。这些安全手段是从互联网中共享信息服务和电子商务的平等交易等的安全需求中假定而来的，其很重要的一个前提是用户不确定，没有一个明确的边界。因此常规的安全手段只能是以共享信息资源为中心，在外围对非法用户和越权访问进行封堵，以达到防止外部攻击的目的，而对共享源的访问者源端不加控制，加之操作系统的不安全导致应用系统的各种漏洞层出不穷，无法从根本上解决。随着恶意用户的手段越来越高明，防护者只能把防火墙越砌越高、入侵检测越做越复杂、恶意代码库越做越大。误报率也随之增多，使得安全的投入不断增加，维护与管理变得更加复杂和难以实施，信息系统的使用效率大大降低。反思上述的做法为老三样：堵漏洞、作高墙、防外攻。其结果是防不胜防。产生这种局面的主要原因是不去控制发生不安全问题的根源，而在外围进行封堵。事实上，所有的入侵攻击都是从计算机终端上发起的，黑客利用被攻击系统的漏洞窃取超级用户权限，肆意进行破坏。注入病毒也是从终端发起的，病毒程序利用计算机操作系统对执行代码不检查一致性的弱点，将病毒代码嵌入到执行代码程序，实现病毒的传播。更为严重的是对合法的用户没有进行严格的访问控制，可以进行越权访问，造成不安全事故。其实现在的不安全问题都是 PC 结构和操作系统不安全引起的。如果从终端操作平台实施高等级防范，这些不安全因素将从终端源头被控制。这种情况在工作流程相对固定的生产系统显得更为重要而可行。以我国电子政务网为例，由政务内网和政务外网两部分组成。政务内网是涉密网，处理涉及国家秘密的事务。政务外网是非涉密网，是政府的业务专网，主要运营政府部门面向社会的专业性服务和不需要在

内网运行的业务。政务内网与政务外网物理隔离，政务外网与互联网逻辑隔离。在电子政务的内外网中，要处理的工作流程都是预先设计好的，操作使用的角色是确定的，应用范围和边界都是明确的。这类工作流程相对固定的生产系统与互联网是有隔离措施的，外部网络的用户很难侵入到内部网络。据2002年美国FBI统计，83%的信息安全事故为内部人员或内外勾结所为，而且呈上升趋势。因此应该以防内为主、内外兼防、狠抓终端源头安全的模式，构筑全面高效的安全防护系统。应该追源头、练内功、控使用、防内患、积极防御。

2. 可信计算技术

近年来，终端安全的思想正在逐渐被人们所重视，对其研究也备受关注。在此背景下，于1999年，包括HP、Intel、Microsoft、IBM等在内的业界几家大公司成立了可信计算平台联盟（Trusted Computing Platform Alliance，TCPA），并于2003年更名为可信计算组织（Trusted Computing Group，TCG）。TCG提出的可信计算（Trusted Computing，TC）概念，其思路就是从终端安全入手，定义未来终端上的一种可信计算环境，通过建立这种可信计算环境来提供各种安全操作功能，达到提高终端安全性的目的。

可信计算尽管是以一种工业规范的形式提出，但其思想却具有普遍而又深远的意义，实际上它是对安全问题的本质回归，使人们将解决信息安全问题的思路转移到解决终端安全问题上来。

可信计算平台是系统安全的必要条件(图14-8介绍了可信计算平台通用结构)。可信计算平台基于可信平台模块（TPM）(图14-9介绍了TPM的内部结构)，以密码技术为支持、安全操作系统为核心。在通用计算机功能的基础上，赋予计算机开机过程中的身份认证、系统资源完整性校验及数字签名/验证、数字加/解密、外部设备的安全控制和日志审计等可信平台的安全功能。可广泛用于军队、公安、安全等涉密机关以及对信息安全敏感的普通商业用户，如金融、电信、电力等行业，有效防御来自外部和内部人员的泄密和恶意破坏行为，特别是对专用网，防止内外勾结攻击显得更为重要。

可信计算平台具有以下几方面功能：可以确保用户唯一身份、权限，工作空间的完整性和可用性；可以确保存储、处理、传输的机密性与完整性；确保硬件环境配置、操作系统内核、服务及应用程序的完整性；确保密钥操作和存储的安全；确保系统具有免疫能力，从根本上阻止病毒和黑客等软件攻击。

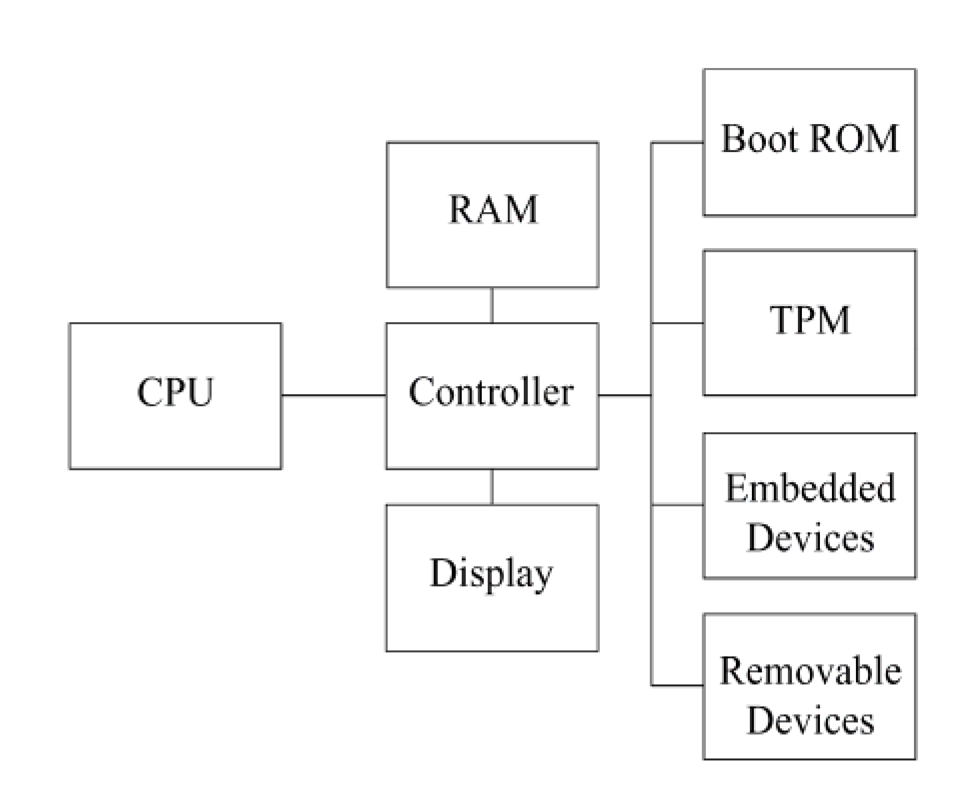


图14-8　可信计算平台通用结构

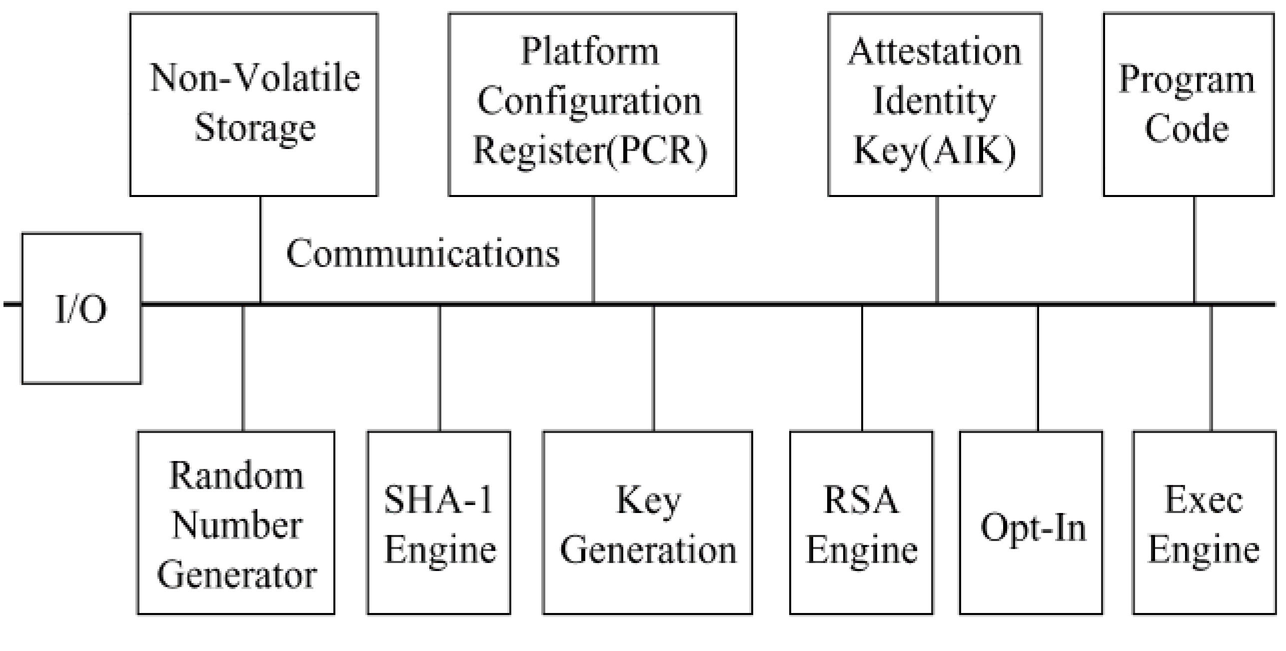


图 14-9　TPM 的内部结构

参 考 文 献

[1] R. L. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21 (2), 1978

[2] Christian Kreibich, Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. ACM SIGCOMM Computer Communication Review, 2004

[3] Lance Spitzner. Honeypots: Catching the insider threat. Computer Security Applications Conference, 2003, Proceedings. 19th Annual

[4] David Moore, Colleen Shannon, Douglas J. Brown, etc. Inferring Internet denial-of-service activity. ACM Transactions on Computer Systems (TOCS), 2006

[5] Treshansky Allyn, McGraw Robert. An overview of clustering algorithms. Proceedings of SPIE-The International Society for Optical Engineering [C], 2001

[6] Lee Garber. Denial-of-service attacks rip the Internet. Computer, 2000

[7] James F. Kurose, Keith W. Ross. Computer Networking: A Top-Down Approach Featuring the Internet (3rd Edition). Addison Wesley, 2005

[8] W. Richard Stevens. TCP/IP Illustrated Volume 1: The Protocols. Addison Wesley, 1996

[9] Klaus Wehrle, etc. The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel. Prentice Hall, 2004

[10] Christian Benvenuti. Understanding Linux Network Internals. O'reilly Media Inc, 2005

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms (2nd Edition). MIT Press, 2001

[12] Alfred J. Menzes, Paul C. van Oorschot, Scott A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996

[13] Atul Kahate. Cryptography And Network Security (2nd Edition). McGraw-Hill, 2008

[14] Charlie Kaufman, Radia Perlman, Mike Speciner. Network Security: Private Communication in a PUBLIC World. Prentice-Hall, 2002

[15] Lance Spitzner. Honeypots: tracking hackers. Addison-Wesley Professional, 2003

[16] Gary Halleen, Greg Kellogg. Security Monitoring with Cisco Security MARS. Cisco Press, 2007

[17] Eric S. Raymond. The Art of UNIX Programming. Addison-Wesley, 2005

[18] Data Encryption Standard. http://en.wikipedia.org/wiki/Data_Encryption_Standard
[19] Public-key cryptography. http://en.wikipedia.org/wiki/Public-key_cryptography
[20] tcpdump/libpcap. http://www.tcpdump.org/
[21] ebtables. http://ebtables.sourceforge.net/
[22] netfilter. http://www.netfilter.org/
[23] OpenSSL. http://www.openssl.org/
[24] Nmap. http://nmap.org/
[25] Snort. http://www.snort.org/
[26] Denial-of-service attack. http://en.wikipedia.org/wiki/Denial-of-service_attack
[27] Sebek：the honeypot project. http://www.honeynet.org/project/sebek/
[28] Boyer-Moore string search algorithm.
http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm
[29] Linux netfilter Hacking HOWTO.
http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html
[30] Oskar Andreasson. Iptables Tutorial 1.2.2.
http://iptables-tutorial.frozentux.net/iptables-tutorial.html
[31] Beej's Guide to Network Programming Using Internet Sockets. http://beej.us/guide/bgnet/
[32] Packet Capture With libpcap and other Low Level Network Tricks.
http://profile.iiita.ac.in/sgarg_02/packet/section1.html
[33] Miguel Rio etc. A Map of the Networking Code in Linux Kernel 2.4.20.
http://www.labunix.uqam.ca/～jpmf/papers/tr-datatag-2004-1.pdf
[34] Alisa Shevchenko. The evolution of selfdefense technologies in malware. Virus analyst，Kaspersky Lab，2007. http：//www.viruslist.com/en/analysis?pubid=204791949
[35] 张绍兰，邢国波，杨义先. 对 MD5 的改进及其安全性分析. 计算机应用，2009 年 04 期
[36] 郑光明，胡博. 基于 MD5 的文件完整性检测软件设计. 湖南理工学院学报，2007 年 01 期
[37] 蒋建春，马恒太. 网络安全入侵检测：研究综述. 软件学报，2000 年
[38] 杨善林，李永森，胡笑旋，潘若愚. K-Means 算法中的 K 值优化问题研究. 系统工程理论与实践，2006 年 02 期
[39] 李军丽. 特洛伊木马病毒的隐藏技术. 网络安全技术与应用，2008 年 01 期
[40] 康治平，向宏. 特洛伊木马隐藏技术研究及实践. 计算机工程与应用，2006 年 09 期
[41] 张新宇，卿斯汉 等. 特洛伊木马隐藏技术研究. 通信学报，2004 年 07 期
[42] 张健. 恶意代码危害性评估标准和检测技术的研究. [博士学位论文]. 天津：南开大学，2009
[43] Schneier Bruce 著. 吴世忠 译. 应用密码学：协议、算法与 C 源程序. 北京：机械工业出版社，2000
[44] William Stallings 著. 孟庆树 等译. 密码编码学与网络安全：原理与实践(第 4 版). 北京：电子工业出版社，2006
[45] Brian W. Kernighan，Dennis M. Ritchie 著. 徐宝文 等译. C 程序设计语言(第 2 版). 北京：机械工业出版社，2006
[46] Daniel P. Bovet，Marco Cesati 著. 陈莉君，张琼声，张宏伟 译. 深入理解 Linux 内核. 北京：中国电力出版社，2007
[47] Michael Beck 等著. 张瑜，杨继萍 译. Linux 内核编程指南(第 3 版). 北京：清华大学出版社，2004
[48] Neil Matthew，Richard Stones 著. 陈健，宋健建 译. Linux 程序设计(第 3 版). 北京：人民邮电出版社，2007
[49] Michael K. Johnson，Erik W. Troan 著. 武延军，郭松柳 译. Linux 应用程序开发(第 2 版). 北京：

电子工业出版社，2005
[50] Jonathan Corbet 等著，魏永明 等译，Linux 设备驱动程序(第 3 版). 北京：中国电力出版社，2006
[51] W. Richard Stevens，Stephen A. Rago 著. 尤晋元，张亚英，戚正伟 译. UNIX 环境高级编程(第 2 版). 北京：人民邮电出版社，2006
[52] W. Richard Stevens 等著. 杨继张 译. UNIX 网络编程. 北京：清华大学出版社，2006
[53] Michael Rash 著. 陈健 译. Linux 防火墙. 北京：人民邮电出版社，2009
[54] Steve Suehring，Robert L. Ziegler 著. 何泾沙 等译，Linux 防火墙(原书第 3 版). 北京：机械工业出版社，2006
[55] Stuart McClure，Joel Scambray，George Kurtz 著. 王吉军，张玉亭，周维续 译. 黑客大曝光——网络安全机密与解决方案(第 5 版). 北京：清华大学出版社，2006
[56] Susan Young，Dave Aitel 著. 吴世忠，郭涛，李斌，宋晓龙 等译. 黑客防范手册. 北京：机械工业出版社，2006
[57] James Stanger Patrick T. Lane 著. 钟日红，宋建才 等译. Linux 黑客防范：开放源代码安全指南. 北京：机械工业出版社，2002
[58] Yusuf Bhaiji 著. 罗进文，王喆，张媛，饶俊 译. 网络安全技术与解决方案. 北京：人民邮电出版社，2009
[59] Brian Caswell，Jay Beale，James C. Foster，Jeffrey Posluns 著. 宋劲松 译. Snort 2.0 入侵检测. 北京：国防工业出版社，2004
[60] Jiawei Han，Micheline Kamber 著. 范明，孟小峰 译. 数据挖掘：概念与技术. 北京：机械工业出版社，2007
[61] Andrew S. Tanenbaum 著. 陈向群，马洪兵 译. 现代操作系统(第 2 版). 北京：机械工业出版社，2005
[62] 杨义先，钮心忻. 应用密码学. 北京：北京邮电大学出版社，2005
[63] 王衍波，薛通. 应用密码学. 北京：机械工业出版社，2003
[64] 卿斯汉. 密码学与计算机网络安全. 北京：清华大学出版社，2001
[65] 卢开澄. 计算机密码学：计算机网络中的数据保密安全(第 3 版). 北京：清华大学出版社，2003
[66] 王石. 局域网安全与攻防-基于 Sniffer Pro 实现. 北京：电子工业出版社，2006
[67] 李善平等. Linux 内核 2.4 版源代码分析大全. 北京：机械工业出版社，2002
[68] 林宇，郭凌云. Linux 网络编程. 北京：人民邮电出版社，2001
[69] 张斌. Linux 网络编程. 北京：清华大学出版社，2000
[70] 杜华. Linux 编程技术详解. 北京：机械工业出版社，2007
[71] 张炯. UNIX 网络编程：实用技术与实例分析. 北京：清华大学出版社，2002
[72] 倪继利. Linux 安全体系分析与编程. 北京：电子工业出版社，2007
[73] 赵炯. Linux 内核完全注释. 北京：机械工业出版社，2004
[74] 毛德操，胡希明. Linux 内核源代码情景分析(上册). 浙江：浙江大学出版社，2001
[75] 恒逸资讯，陈勇勋. 更安全的 Linux 网络. 北京：电子工业出版社，2009
[76] 刘文涛. 网络安全开发包详解. 北京：电子工业出版社，2005
[77] 许治坤，王伟，郭添森，杨冀龙. 网络渗透技术. 北京：电子工业出版社，2005
[78] 曹元大，薛静峰，祝烈煌，阎慧. 入侵检测技术. 北京：人民邮电出版社，2007
[79] 阎慧，王伟，宁宇鹏. 防火墙原理与技术. 北京：机械工业出版社，2004
[80] 张建忠，徐敬东. 计算机网络实验指导书. 北京：清华大学出版社，2005
[81] 吴功宜. 计算机网络高级教程. 北京：清华大学出版社，2007
[82] 吴功宜等. 计算机网络高级软件编程技术. 北京：清华大学出版社，2008

[83] VC 知识库. http://www.vckbase.com/
[84] 中国 OpenSSL 专业论坛. http://www.openssl.cn/
[85] 黄一文. Linux 路由表的结构与算法分析.
http://blog.csdn.net/rwen2012/archive/2007/10/05/1811578.aspx
[86] 杨沙洲. Linux Netfilter 实现机制和扩展技术.
http://www.ibm.com/developerworks/cn/linux/l-ntflt/
[87] Linux 高级路由和流量控制 HOWTO 中文版. http://lartc.org/LARTC-zh_CN.GB2312.pdf
[88] 加固 Linux 内核. http://www.chineselinuxuniversity.net/courses/kernel/guides/313.shtml
[89] 张健. 2007 年中国计算机病毒疫情调查技术分析报告，国家计算机病毒应急处理中心，2007
[90] Robert S. Boyer，J Strother Moore. A Fast String Searching Algorithm.
http://www.cs.utexas.edu/users/moore/publications/fstrpos.pdf
[91] 精确字符串匹配的 Boyer-Moore(BM) 算法.
http://hi.baidu.com/tjrac_miracle/blog/item/bd2ad394efb5954dd0135e72.html
[92] Pekka Kilpel äinen. Boyer-Moore Matching. Biosequence Algorithms，Spring 2005
[93] Pekka Kilpel äinen. Set Matching and Aho-Corasick Algorithm. Biosequence Algorithms，Spring 2005
[94] AC 多模匹配算法小结. http://blog.csdn.net/lindan1984/archive/2006/12/20/1450307.aspx
[95] P3Scan. http://p3scan.sourceforge.net/
[96] RFC1321. The MD5 Message-Digest Algorithm. http://www.ietf.org/rfc/rfc1321.txt
[97] RFC4732. Internet Denial-of-Service Considerations. http://tools.ietf.org/html/rfc4732
[98] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification，Microsoft Corporation，Revision 8.0.
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
[99] Clam AntiVirus User Manual. http://www.clamav.net/doc/latest/clamdoc.pdf

读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602 室 计算机与信息分社营销室 收

邮编：100084　　　　电子邮件：jsjjc@tup.tsinghua.edu.cn

电话：010-62770175-4608/4409　　　　邮购电话：010-62786544

教材名称：网络安全高级软件编程技术

ISBN：978-7-302-21904-0

个人资料

姓名：________________ 年龄：_______ 所在院校/专业：_____________________

文化程度：____________ 通信地址：_______________________________________

联系电话：____________ 电子信箱：_______________________________________

您使用本书是作为：□指定教材 □选用教材 □辅导教材 □自学教材

您对本书封面设计的满意度：

□很满意 □满意 □一般 □不满意　改进建议_______________________________

您对本书印刷质量的满意度：

□很满意 □满意 □一般 □不满意　改进建议_______________________________

您对本书的总体满意度：

从语言质量角度看 □很满意 □满意 □一般 □不满意

从科技含量角度看 □很满意 □满意 □一般 □不满意

本书最令您满意的是：

□指导明确 □内容充实 □讲解详尽 □实例丰富

您认为本书在哪些地方应进行修改？（可附页）

____

____

您希望本书在哪些方面进行改进？（可附页）

____

____

电子教案支持

敬爱的教师：

为了配合本课程的教学需要，本教材配有配套的电子教案（素材），有需求的教师可以与我们联系，我们将向使用本教材进行教学的教师免费赠送电子教案（素材），希望有助于教学活动的开展。相关信息请拨打电话 010-62776969 或发送电子邮件至 jsjjc@tup.tsinghua.edu.cn 咨询，也可以到清华大学出版社主页（http://www.tup.com.cn 或 http://www.tup.tsinghua.edu.cn）上查询。